THE mini-tasker

DECUS
RT-11 SIG NEWSLETTER

December 1984 Volume 10-5

SJ RTMON RMON FILEX CSI ODT

PIP LD SYSMAC FB

DIR QUEMAN KMON

QUEUE TECO

PAT DUP

VM

BINCOM KED K52

LIBR DUMP BUP

LINK SIPP

SRCCOM RESORC FORMAT

SLP TTYSET IND

MACRO XM JSW HELP

BNO

Participate in the

DECUS Europe Symposium

Cannes , France

September 16-20, 1985.

See you there !

THE mini• tasker

DECUS
RT-11 SIG NEWSLETTER

December 1984 Volume 10 Number 5

Contributions to the newsletter should be sent to:

Ken Demers
Adaptive Automation, Inc.
5 Science Park
New Haven, CT 06511
203 786-5050

Other communications can be sent to:

John T. Rasted
JTR Associates
58 Rasted Lane
Meriden, CT 06450
203 634-1632

or

RT-11 SIG
C/O DECUS
One Iron Way
MR2-3/E55
Marlboro, MA 01752
617 481-9511 Ext. 4141

The mini-tuker

DECUS

RT-11 SIG NEWSLETTER

December 1984 Volume 10 Number 5

Contributions to the newsletter should be sent to:

Ken Deemer
Adaptive Automation, Inc.
5 Skyline Park
New Haven, CT 06511
203 786-3059

Other communications can be sent to:

John S. Reated
UTM Associates
15 Wasted Lane
Marlboro, CT 06550
203 323-1492

or

RT-11 SIG
D/D DECUS
One Iron Way
MRO-1/E55
Marlboro, MA 01752
617 480-7211 Ext. 6741

Table of Contents

FROM THE EDITOR

After approximately eight years, I have decided to relinquish my position of RT-11 newsletter editor. If you are interested in becoming the next newsletter editor, please contact me as soon as possible. The next editor will be chosen from those applicants that respond by February 28th, 1985. I will continue to be very active within the RT-11 SIG. I hope to become the RT-11 SIG'S communication representative. The publication of the "mini-tasker" is one of the responsibilities of this new position.

I would personnally like to thank all of you around the world, who have taken the time and effort to submit so many informative articles to the "mini-tasker" over the years. Please keep up the good work.

USER INPUT

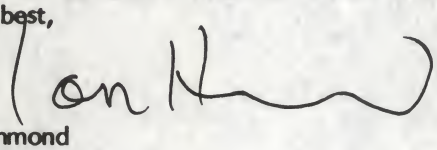
I haven't sent anything for awhile. We have been rather busy here. To make up, I have enclosed some 45 pages of notes I used for European DECUS training sessions the last two years. They are not well coordinated, full of minor errors, and hardly in language intended for publication.

But I think they could be valuable for two reasons: first, because they deal with some monitor issues that I have not yet seen in print. The second reason is precisely because they are not in 'publication language' - they were written in a hurry and seem to me to document fairly well the way programmers talk to each other.

I have held them back because I believe that training session notes belong to the attendees for a year. They pay for them and patrons rules apply. Two years ago I would not have published them anyway - but DEC have loosened up since then and are not as easily offended.

45 pages is a lot - but at least you can't complain about not getting enough from me! I have some more stuff in the pipeline and plan to rework the short history of RT-11 to deal with new releases.

All the best,



Ian Hammond

rt-11 internals

No one can explain 'how an operating system works'. RT-11 probably has around two thousand separate mechanisms - many of which are interdependent. Further, as RT-11, and other DEC operating systems, tend to be innovative, there is no current literature which adequately covers many of the techniques used.

As Ralf Stammerjohn says, an operating system is just a large program. Thus an operating system can be approached with the same techniques you would use to analyse the 'generic' program.

Therefore, the object of todays session is not to provide a detailed description of the whole monitor. The goal of todays session is to focus on the best techniques to use in analysing the RT-11 operating system.

The monitor is more than the sum of its code and its comments. Often it is more important to know 'why' a particular approach is used in the code. And the 'why' often has its roots in history or politics. Thus we will deal today with some history and politics.

RT-11 monitor code is always subject to space restrictions. The monitor and the CUSPS are entirely written in MACRO. Some of the space-saving techniques make the code difficult to follow. We will look at some specific examples here.

During the morning, and perhaps the early afternoon, we will work bottom-up thru Bootstrap, KMON/KMOVLY, USR and RMON.

However, since this session is advertised as 'starting where the SSM leaves off' I have a difficult task - since the SSM is very detailed. Therefore I plan to spend a large part of the afternoon dealing with interrupts, forks, scheduling, context switching, etc. That is, the dynamics of the FB operating system.

The definite non-goals for todays session are XM, MTT and FORTRAN & BASIC. But, I will mention some related material to the XM monitor (I have rarely used the first three of these and I only seem to get a bad reputation for admitting I often use (and like) the last).

This session is a general training session. We will however have trouble covering all the material related to what is a very complex piece of software. Thus, we cannot afford to get sidetracked by individual problems in the monitor. You should see me during the week if you have specific problems, or use the Q & A sessions. We will also try to set up a birds-of-a-feather for monitor internals during this symposium.

I have already presented some of this material in the SIG newsletters and at other DECUS presentations. I cannot apologise for the repetition, since without it this training session would be very incomplete. But, there is a lot of new material as well.

You should have all received handouts and a pocket reference. I want to remind you that this material is copyright - specifically, it should not be reproduced in SIG newsletters. The handouts are working notes: They would give a misleading impression without the accompanying commentary and discussion.

RT-11 HISTORY

An operating system is not much more than the sum of the work that was put into writing it. It reflects both the strengths and the weaknesses of its authors. In the case of RT-11 there a number of points that are worth knowing when looking at the monitor.

DEC's first O/S for the eleven, DOS-11, was not a success. It was too small to be a big system, and too big to be a small system. Around 1970 DEC realized they needed a small, fast, easy-to-use operating system for the eleven, in the same tradition as OS/8 for the PDP-8. And indeed, RT-11 is a child of OS/8.

OS/8

OS/8 is generally credited to Richy Lary. Lary also wrote the micro-code for the LSI-1 and the PDP-11/60.

The first version of RT-11 included members of the OS/8 team.

The entire monitor structure, and most of the original philosphy of RT-11 comes from OS/8: RMON, USR, CSI and KMON and most of the V1 monitor commands.

8KW and DECtapes

RSX-11 was around as a 'run-time' system during the early days of RT-11. One proposal was that RT-11 use a subset of the RSX-11 file structure, and this was investigated. However, the idea was eventually dropped and an OS/8-like structure was adopted. The reason: RT-11 had to fit into 8kw.

This point illustrates one of RT-11 strongest influences: the system was designed to run in 8kw.

The second goal was good performance on DECtapes.

The 8k and DECtape goals influenced the initial design of RT-11 heavily, and the architecture today is mostly unchanged.

VERSION ONE

RT-11 V1 was small, fast but not always easy to use. That is, it used to crash on occasions, and gave very few reasons why.

The error messages were also small, fast and not easy to use:

```
?M-FIL NOT FND?
```

The documentation was small, fast and not always easy to use. The entire system documentation occupied somewhat less space than the current System Users Guide. And about a quarter of that was used for fairly useless appendices.

However, the basic monitor design was clean. This is principally because the design was based on a mature system - in this case OS/8.

Here is a list of the strengths of the system that were already clear with V1:

small	ran in 8kw (now requires 12kw)
fast	very little system monitor overhead
	very fast contiguous file structure
	unmapped environment permits fast interrupts
easy-to-use	small monitor permits large programs
	very simple program request structure
	programs are loaded into low memory
	no sysgen required to run system

The major weakness in V1 was the implementation of program request calls. The V1 documentation warned this would change in V2 (more on this later). Other weaknesses emerged later as the monitor design was expanded.

V2

From our point of view, looking at the monitor, V2 should be considered one the best RT-11 releases. V2 added the FB monitor. To do this the developer had to rebuild the SJ monitor without substantially changing its architecture (for compatibility reasons). Now, since the SJ system was built for a singe-user environment, it did not employ many of the mechanisms that you would expect to find in a multi-user environment.

The original coding of the V2 FB monitor was so solid that the current FB monitor is still mostly the same code. Even though the FB monitor only needed support for two jobs, the developer produced scheduler/system that would handle up to 128 jobs. When it came time to do the System Job monitor (MJ) much of the work had already been done.

Foreground programs are the result of a link operation - no special code is needed in the program. This follows RT-11's approach to overlays, and later in V4 to the virtual settop, which are also LINK operations. FG .REL files are a superset of a BG program. You can RUN a .REL file in the BG.

V3

What can you say about Aversion three. Version three was truly a dramatic release of RT-11. Fantastic and rotten at the same time. RT-11 reached a peak of popularity with V2, and the V3 group were given the go-ahead for a lot of new functionality. They literally shovelled the functionality into the system. And as I have said before, RT-11 got a bit too big for its bootstraps. V3 added:

XM monitor	DCL - abbreviations, factors
Command files	HELP SHOW
Escape Sequence support	Error logging

V3 also added a long list of bugs that are still being discovered. If the system breaks today, then there is a good statistical chance that the problem will be traced to V3.

But, V3 changed the face of RT-11. It was a new system. In retrospect I am very glad that V3 happened; whether Digital would underwrite an effort to add DCL support to RT-11 today is doubtful. But then, Digital's recent more conservative approach to operating system development is partly because of the poor reliability of ventures like V3 which add so much functionality to the system.

Despite the bugs, a lot of the V3 code is very good. The DCL implementation is a miracle in some ways (more later).

V4

V4 put RT-11 back on the tracks; it was RT-11's biggest maintenance release. One of the main tasks of the group was to clean up their own house - i.e. the aftermath of V3. Anton Chernoff, who led V2 and did the FB monitor, returned from a stint with RSTS to do V4. And V4 has prooved to be the most reliable release. You can still use the distribution kit without adding patchs for most tasks. Among the clean-up tasks were:

PIP, DUP rewritten	DIR moved from SUPERMAC to MACRO
HELP moved from TECO to MACRO	SYSGEN moved from FORTRAN to MACRO
Error logging redone	Escape sequences removed
XM substantially debugged	XMSUBS and MTT created
Monitor/handler separation	

V4 was very compatible with V3. Few users would have had troubles moving to V4. V4 kept the changes in the background, quiet and subtle. Things like automatic installation of handlers; filenames/devices added to error messages; virtual settop. V4 added the System Job monitor.

V3 failed to deliver a software support manual (who could describe V3?); V4 more than made up for this with an excellent SSM that made operating system tasks that were usually reserved for wizards, available to all advanced programmers.

TERMINOLOGY

MACRO and SHAKESPEARE

Is it MONLOW or SYSPTR? A rose is not a rose by any other name to MACRO - it is an undefined symbol. RT-11 did not define standard mnemonics for many of the monitor structures until fairly recently, and even now there does not exist a file which defines these for users. I have included a pocket reference with most of the main definitions.

KW

'K' defaults to kilowords, not kilobytes.

USR

USR is pronounced 'U. S. R.', not 'USER', which is ambiguous with that mystic object, the user. If you have read the symposium schedule, you will see that the U.S.R. has also become ambiguous - see the session titled 'RT-11 Tutorial of Device Directories and the USSR'. (This session will deal with file-structure internals - so I will leave that topic alone today)

MJ

There is no well defined mnemonic for the System Job monitor, and SJ won't work. I will call the System Job monitor 'MJ' - for Multi-Job, and 'XJ' for XM System Job monitor.

BG

Although the BG program really only exists in a FB/XM/XJ environment, I will also refer to the program area of the SJ monitor as the BG. This makes sense if you think of the SJ monitor as a 'BG only' monitor.

EXACT NUMBERS

During todays session I will refer to a number of mechanisms you can use with RT that are not documented. Whether or not you can usefully employ these mechanisms depends on your understanding of them. Therefore, I will describe them, but I will not provide recipes for their use. Further, you should be aware that since they are not documented, they are subject to change. Indeed, some of them are certain to change with V5.

SLIDES

Rather than invest my time in making set of slides, I have put together this set of fairly detailed notes on the session. Thus, we will be working mostly from the notes rather than from the non-existant slides. The advantage is that you will have to do less in the way of actually taking notes. The disadvantage is that you will spend more time following notes than looking at me - well, maybe that is an advantage.

10 O'CLOCK

I rarely say anything sensible before the hour of 10 AM. So forgive me if I take a while to get started. This is also the reason that I have reserved the first hour or so for the non-technical topics of history and politics.

RT-11 POLITICS

Why are history, philosophy (in its technical sense) and politics important in understanding the internals of an operating system?

An operating system is the sum of the work of its authors. Understanding the limitations of, and the pressures on, the development group is important to understanding the software. The answer to the question 'why is this piece of code like this' is often only answered by a historical, philosophic or political reason. A full understanding of RT-11 is not possible without touching on these topics.

All the opinions expressed below are opinions - based on guesswork and a long history of DEC watching. There are certainly mistakes - so please don't quote me. This section is intended to give an overall impression only.

WORKLOAD

Typically, the development group consists of up to seven programmers and up to five or six technical writers. To this you must add two or three managers. The RT-11 group not only handles the development of new versions, they also handle maintenance of the existing version (and perhaps previous versions).

The RT-11 group are not responsible for languages such as FORTRAN and BASIC. (I am not sure if anyone is presently responsible for BASIC!) Some of the RT-11 components are shared with other operating systems. MACRO is much the same for RT-11 and RSX-11. RT-11 produces KED, but the same basic KED software is also used for RSX.

EXTERNAL INFLUENCES

A development group inside a company as large as DEC is subject to the corporate politics. The company defines and maintains goals for the various operating systems. As we have often heard from Digital:

> RT-11 is Digitals small, fast, single-user system
> RSX-11M is Digitals multi-task, multi-user system
> RSTS is Digitals multi-user time-share system

Implicit in these statements, for example, is that RT-11 is not permitted to become a multi-user system.

RT-11 must also follow corporate philosophy in a positive sense. This is the reason that RT-11 V3 adopted escape-sequence support and error-logging.

RT-11 is also subject to the pressures of other project groups within Digital. This is the main reason that RT-11/XM arrived.

RT-11 is also subject to user pressures. The group solicits and maintains a long list of user wishlist items. These are regularly discussed, and many of them appear in new versions. The reason that some do not are because of the influences mentioned above, and because of the following:

RT-11 INTERNAL INFLUENCES

The development group has finite resources in terms of programmers, writers and time. There is never time to do everything. The complete wishlist listing is more than a couple of inches high.

Compatibility with previous releases is a large issue. With over 50,000 RT-11 sites active, the group must be extraordinarily careful about changes to the system. This affects monitor components the most.

Some RT-11 structures have definite limits that cannot be exceeded. For example, V1 of RT-11 defined the USR at 2kw. This limit may not be exceeded. The 2kw limit was met some years ago, and any new USR functionality requires finding new space by compressing the code. At present it requires around about 2 days work to find an additional word of space in the USR.

The development group also have their own wishlist items.

POLITICS

An operating system is not something that will be supported and developed ad infinitum. A development group like RT-11 is an expensive proposition. Therefore the future of RT-11 is dependent on Digital policy - which translates into politics as in any large company.

During V3 RT-11 was at its zenith inside Digital. Shortly after the release of V4 there was talk of its demise. This seems to have changed in the past six months (which perhaps has something to do with the slide from 36% growth to 26%). At present RT-11 seems very healthy.

RT-11 is Digitals biggest selling system in terms of numbers. Over 50,000 sites have been reported. Obviously, this is a profitable product for the company. Further, RT-11 is an industry standard for OEMS and system houses, and for many of Digitals own layered products.

However, 'when your on a good thing stick to it' also applies. One reason that RT-11 is so popular is because it is so simple. Thus, its popularity does not ensure that the system will grow and grow. The reverse is true.

Questions like 'Will RT-11 run on the professional' can no longer be asked within the same frame of reference we used to use in the RT-11 vs RSX-11 wars. The Professional is a new kind of product aimed at a new kind of market. It is not aimed at the professional programmer, not even at the amateur programmer. It is aimed at the absolute non-programmer who wants to know nothing about the internal operations of the machine.

In a few years we will find the Digital user community divided into producers and consumers. The Professional is aimed at the straight consumer. 'The Professional' is not the end of the line, it is the beginning of a new line. I guess it will eventually steal a lot of layered products from RT-11.

HUMAN WEAKNESS

RT-11 is the product of human beings and reflects the strengths and fallibility of that species. The RT-11 monitor code is the produce of four different development groups - not all of whom understood completely what the code produced by the preceding group was supposed to be doing.

The original code is often elegant, and almost without exception professional. The patched code is usually spaghetti. Snakes and ladders. And it has little bugs and big bugs in it. This problem is not unique to RT-11. Something was rotten in Denmark a long time ago. As one writer points out, the first machine, the Babbage machine, was also the first machine to be delivered late and not work.

This is where history becomes important. RT-11 V3 added the most functionality to the system. It also added the most bugs. When you are looking at the code you should treat V3 additions quite differently to V2 and V4.

RT-11 MEMORY LAYOUT
RT-11 looks like this:

At this point the author produces a strange object 1mm wide by 2 meters high.

The basic PDP-11 memory consists of 32k 16-bit words. That is a structure that is 2048 times higher than it is wide (if a bit is symmetric).

4kw (1/8th) of the memory is allocated for I/O space. This may be reduced to 2k (1/16th) with a 30kw memory switch. Less than 256w (1/128th) is allocated for vectors. In retrospect the vector space is far too small (less than 80 vectors).

RT-11 requires at least 2kw (1/16) for the SJ monitor. An FB system with the USR set NOSWAP requires 6kw (3/16).

Using this model, a 128kw machine is eight meters high. A 1 megabyte machine is 32 meters, and a 4 megabyte machine reachs 96 meters. My 30 megabyte disk drive would be about a kilometer long (if these figures are incorrect then I have proved yet once again that I am not capable of the simplist arithmetic (particularly if it involves division which I no longer understand at all)).

When RT-11 bootstraps, it loads resident components to the top of memory. The system device handler goes at the very top.

Below that comes RMON - the Resident MONitor.

```
                i/o space
28kw/30kw
                sy.sys
                RMON
```

Below RMON is some plastic space that stretchs to fit in LOADed handlers, FG or System jobs and command files (during their exectution). We will call this the 'Stretch Area'.

```
                i/o space
                sy.sys
                rmon
                LP.SYS
                MYFG.REL
                MYCOM.COM
```

If the USR is set NOSWAP, it is loaded under the stretch area.

When KMON runs, it is loaded just below the USR. KMON includes the 512 word overlay section (KMOVLY). Nothing fits between KMON and the USR.

```
                i/o space
                sy.sys
                rmon
                lp.sys
                myfg.rel
                mycom.com
                USR
                kmovly
                KMON
                MYPROG.SAV
                vectors
```

And if your program is small enough, it will load into memory without swapping KMON out.

RT-11 is the DEC system for unmapped machines.

BOOTSTRAP

"Just because everything is different doesn't mean that anything has changed."
San Francisco Oracle

"Once in my life I would like to own something outright before its broken."
Miller, Death of a Salesman

(These quotes are comments to the patch levels recorded in BSTRAP)

The bootstrap is RT-11's cellar (dungeon?). This code occupies the first five blocks of the monitor image and resides on blocks 0 and 2 thru 5 of a bootable device.

Starting the Machine presents its own special problems. Its a little bit like the reverse regurgitation of the common cartoon of a small fish eating a slightly smaller fish only to be eaten itself by a slightly larger fish, ad infinitum.

(1) The SSM tables 7-6 and 7-7 give the wrong locations for the date and time. The correct locations are given below. There is a summary of the all the boot fixed locations in the pocket reference.

```
5000 btime   ; high-order time from dup
5002         ; low-order time
5004         ; date
```

(2) If you want to write a program that bootstraps a device you should use the following algorithm:

1. Get and assemble the bootstrap somewhere in memory
2. Do a hard RESET instruction
3. Read 1 word from the device with a .READ
4. Go to PR7
5. Copy the boot into low memory
6. Setup the registers
7. Jump the the bootstrap

It is crucial that you reaccess the device after the RESET. This emulates hard bootstraps that also access the device after the RESET.

STARTING UP

One of the first things RT has to do is work out who its parents are. The purpose of this genetic search is to work out whether it has inherited such things as a PS or MTPS/MFPS instead. RT-11 uses the following tests:

```
177776 PS       PSW
177540 LKCS     Line-clock status register
1      CLOCK$   Processor has a clock if it has LKCS or no odd-address trap
CFCC   FPU$     FPU processor

VT.CSR VS60
172540 PCLOCK
(r5)   ^rPD     If its a PDT then VT11 and VT.CSR do not exist
172032 VS6.SP   VS60
177760 P7.LSS   11/70
MUL 3,3 EIS
177570 SR       Read checks for switch register
177570 SR       Write checks for lights register
MEDINS,100      11/60 check
CMPN   CIS      Commercial Instruction Set
```

SYSGEN parameters
One of the undocumented SYSGEN parameters in RT-11 is MEX$T which encloses code for 22-bit support. However, this does not mean that 22-bit support is already there in RT-11; this code is just in preparation for that task.

System handler
The system device handler goes right at the top of memory. It is important that system device handlers keep all memory references inside their address spaces - otherwise they risk hitting No Mans Land in the I/O page.

HANDLER INSTALLATION
One of the nicest things RT-11 does for you is automatically install all the handlers it finds on the system device. It makes sure that these handlers are valid by calling the installation check routine at location 200 of the handler, if such a routine exists.

Note that the bootstrap process calls the installation check at location 202 for a handler that is being loaded as the system device handler.

A problem arises if you have more handlers on the system device than you have device slots for. For example, I know of sites that may have up to 40 handlers on SY:. In this case you would like the automatic bootstrap load to be more selective. The following code will stop RT-11 from automatically installing a handler at boot time. However, you can still install the handler with the INSTALL command.

```
.asect
.=200
        br      check
        return                          ;always accept load as SY:
check:  cmp     (sp),#6000              ;is this coming from BSTRAP?
        return                          ;c=1 => it came from BSTRAP

        r patch                         ! the same with patch
        dd.sys
        200/    0       401
        202/    0       207
        204/    0       21627
        206/    0       6000
        210     0       207
        ^C
```

VM: Installation
One of the most clever pieces of code in RT-11 is the VM: handler installation code. The size of the VM: device depends on how much memory your computer has. VM: does not know this before the installation - so it cannot set location 50 of the handler to tell RT-11 the device size in blocks.

But it can. RT-11 runs the installation check before picking up the device size from location 50. The VM: handler installation routine sizes memory and moves the result to relative location 50 of the handler in memory. RT-11 then picks up the device size.

KMON

I call KMON the Kaleidescope Monitor for two reasons. First, because of the wide range of functions it provides. Second, because its environment is like the moving patterns of a Kaliedescope: KMON is not only position independent, it is the only piece of software I know that slides up & down in memory as it executes.

Add to the these two features the fact that KMON includes an interpreter for one of the worlds more difficult and idiosyncratic syntaxes, DCL, and you have the reason why KMON is the most complex part of the monitor.

KMON is also subject to space restrictions. For example, the undocumented PASCAL and TECO commands are available only in the FB/XM monitors - there was not enough space left in SJ monitor for these commands.

KMON has the following general layout:

USR
KMOVLY
KMON
relocation
move routines

USR - 2kw

It is important to realize that the USR is always contiguous with KMON. Thus, KMON can directly address locations in the USR. When KMON slides, the USR slides with it. (XM is different)

KMOVLY - 0.5k

KMOVLY is a 512. word overlay section. KMOVLY is not just an add-on to KMON: This is where most of the work is actually done. KMON acts largely as a set of service routines for KMOVLY procedures.

(In earlier releases the SJ KMOVLY was 256. words to save space).

KMOVLY routines include everything from APL to UNLOAD.

A KMON overlay may not exceed 512. words. Now, to keep the block size of KMON down to a minimum, there is also an effort to get as much code as possible into each overlay. Thus, there is often some otherwise inexplicable juggling of routines between overlay segments.

Further, KMOVLY routines may not perform certain functions. For example, they must return to the KMON root segment to slide up and down. Thus KMOVLY routines are often interrupted with seemingly meaningless jumps to short KMON routines.

KMOVLY does not use the standard RT-11 overlays produced by the linker. The overlays are defined largely by a set of MACRO's.

KMOVLY always runs with the USR locked. This means it is free to use the USR buffer for I/O operations (which it does for LOAD, SET etc.).

KMON - 3.5kw

The KMON root segment is the heart of the DCL interpreter. It also handles most of the functions concerned with exiting a BG program and R'ing a BG program.

KUMOVE:

"The awful shadow of some unseen power, floats 'tho unseen, amongst us"
- Shelly, Hymn to intellectual beauty

The quotations in these notes come from the RT -11 sources at the label shown above (in this case KUMOVE). The quotes started in the FB monitor of V2. They are often a very good indication of the kind of problem faced by the code they describe.

The first few routines are used to move KMON/USR up and down in memory.

There is one routine that cannot be moved by the move routine, - and that is the short routine that actually does the moving. This is like standing on the branch of a tree that you are currently chopping off. The size of the routine that cannot be moved sets the size for the minimal move in memory. This is currently 17. words.

The move-up routine is located at the start of KMON.

During execution, the code that moves KMON/USR down is located, appropriately, at the top of the USR.

movedn
USR
KMOVLY
KMON
moveup

However, as we all know, the USR is very tight on space. Therefore the MOVEDN routine is actually stored in KMON, and moved into an impure area at the end of the USR before execution.

MINMOV
This size of the Minimal Move is important - since it sets the basic unit size with which KMON can allocate memory.

KMON can only free a segment of memory that is at least twice the size of MINMOV (this parameter is called MINFRE).

WHEN DOES KMON MOVE
KMON slides to make space, or reclaim space. Initially the picture is:

RMON
USR
KMON

To LOAD a handler, KMON slides down to get the space:

RMON
handler
USR
KMON

After an UNLOAD it slides up again.

The following components require space:

Command files
Handlers
Foreground or System Jobs
GT handler

OPTIMISING SPACE ALLOCATION

Assume the following commands:

commands	memory
	RMON
LOAD MT	MT
FRUN MYJOB	MYJOB.REL
LOAD VM	VM
	USR
	KMON

Now, unloading MT leaves a hole in memory:

commands	memory
	RMON
UNLOAD MT	<hole left by MT>
	MYJOB.REL
	VM
	USR
	KMON

But, unloading VM lets KMON reclaim the space:

commands	memory
	RMON
	<hole left by MT>
	MYJOB.REL
UNLOAD VM	USR
	KMON

Thus, the rule is, LOAD the things you need least, last. That way, if you need some additional memory for a job, you can unload some handlers to get the space, without having to unload everything.

Lets reLOAD the VM handler:

commands	memory
	RMON
	<remainder of MT hole>
LOAD VM	VM
	MYJOB.REL
	USR
	KMON

KMON searchs thru the memory map to find a space large enough for the handler. Only when it finds no such space does it move down KMON/USR to make space.

Now, lets reLOAD MT:

commands	memory
	RMON
	<remainder of MT hole>
	VM
	MYJOB.REL
LOAD MT	MT
	USR
	KMON

Using this kind of technique, you can very easily use up all memory.

COMMAND FILE ALLOCATION

Command files are usually a special case in every sense, and this also applies to memory allocation.

KMON allocates space in the stretch area to store command file data.

KMON reads a command file until it finds EOF, or CTRL/C.

KMON works a line at a time. It reads from the command file into a line buffer, stripping comments.

Each line is stored separately in the stretch area. This means, that the monitor memory allocation routine is called each time it needs to store data in the stretch area.

Now, consider the following command file:

```
TIME
TIME
...     <1000 more TIME commands>
TIME
TIME
```

KMON will allocate memory separately for each line, moving down in memory when a hole is not available.

Now, since MINMOV sets the size of the minimum hole, and MINMOV is 17 words, KMON will slide down in memory every 34 characters. Thus, since KMON+USR is about 6k words, and the move loop is about 6 instructions, it will require about 1000 instructions to store each byte.

The command file above takes many minutes before it displays the time.

COMMAND FILE HOLES

One good way to waste space is with command files that load handlers.

Consider what happens when weplace the first set of LOAD commands in a command file:

commands	memory
	RMON
@MYCOM	MYCOM.COM
LOAD MT	MT
FRUN MYJOB	MYJOB.REL
LOAD VM	VM
	USR
	KMON

After execution, the memory map will be:

commands	memory
	RMON
	<hole left by mycom.com>
	MT
	MYJOB.REL
	VM
	USR
	KMON

RULE

If you are tight on memory space, interpolate CTRL/C's between command file lines.

KMON START

Having found a reasonable place in memory to execute, KMON generally starts by finishing up the EXIT from the program that just completed.

This piece of code has an average of one conditional every three lines, which makes it difficult to read. But, there is not much happening there anyway.

Two points of interest are the CONTU code (which is used for MINC) and the IND code which is used for AUTOPATCH. However, the essential point for IND is the operation of the INDEXT monitor offset, which we will deal with later.

LNK$IF

Ignore all the code surrounded by LNK$IF conditionals. This code has not been fully finished - it is the remains of the end of V3. This code was to have handled overlay command files for the LINK command.

R & RUN

R, RUN and GET have a significant amount of code in the KMON root.

The difference between R and RUN is as follows:

R

R uses USERTOP (location 50) to determine how long the program in. An R command requires two read operations. The first gets block 0 and acquires USERTOP. Block 0 is moved down to the vector area, and then the resident monitor is called to issue the second read request which pulls in the remainder of the program.

RUN

RUN uses a scatter load. After acquiring block zero, RUN uses the bitmap (locations 360:400) to work out which blocks to load into memory. Thus, blocks that we not written into by the linker will not be loaded into memory. This feature was primarily designed for use with the GET & SAVE commands, to permit you to build program images in memory, with multiple GET's, and then store the result with a SAVE command. These commands come directly from OS/8, where this kind of technique was used in place of a LINKER. (They are redundant in RT-11).

Note, that KMON may occupy memory that will be needed by the program when it is eventually loaded. Thus, RUN copies such code out to the swap file. Then, the program is loaded into memory by swapping in from the swap file.

MRRT simplifies this whole procedure by locating all the code that runs a program in the resident monitor. Thus, swap files are not required.

SJ SRESET

Another way that space was saved in USR was by moving the .SRESET routine in to the USR buffer area. Thus, everytime a program executes a .SRESET request under the SJ monitor, it causes a reread of the USR. This is kinda crazy (note that BASIC and KED perform three .SRESET's on the way up - which causes three extra clacks on a floppy drive as they reread the USR).

Since this is crazy, and the monitor people knew about it they put in a work-around for KMON. KMON has a second copy of the .SRESET code that it uses instead of running the USR version.

Note, that PIP, DIR and DUP do not use .SRESET either (because of efficiency, and critically for /WAIT operations).

sj sreset

You can do a soft reset a la PIP with the following skeleton:

```
        <purge all channels, except overlay channel if in use>
        mov     @#sysptr,r0     ;point to the monitor
        add     pnptr(r0),r0    ;point to the pnames
10$:    tst     (r1)            ;this one loaded?
        beq     20$             ;nope
        .releas r1              ;yes - forget it
        bcs     30$             ;that was the last one
        tst     (r1)+           ;do the next
        br      10$             ;
30$:                            ;
```

Note, that this routine does not reset CDFN or QSET.

EXIT

While we're on the subject, we will finish up the discussion on KMON swapping.

If you exit a program, RMON has to find space in memory to run KMON. KMON always runs as high as possible in memory. If your program is small, it will remain in memory and KMON will be loaded above it.

Indeed, if you did not use .SETTOP to acquire the memory used by KMON it will still be in memory and will not need to be reread.

However, if your program overlays the area needed by KMON, then that portion of your program will be written out to the swap file (SWAP.SYS), and then KMON will be read into that area.

RT-11 uses this technique so that the following commands will work:

1. REENTER
2. SAVE
3. Examine, Deposit and Base

These commands are typically rarely used. Thus, for the most part all this swapping is redundant.

Indeed, the idea behind MRRT was to get rid of this swapping, since it made TU58's more impossible than they already were.

You can stop KMON swapping your program out with the following command at the end of your program:

```
        .settop #0              ;forget the memory we got
```

SWAPPING & SLIDING

Before leaving sliding and swapping we should ask the question Why? Why Slide?

I am sure that if RT-11 had been designed in 1980 rather than 1970 this kind of sliding would not have been used. The original reason was the 8kw/DECtape goal. Both those restrictions dissappeared years ago. The design of KMON is now archaic. A pain.

It should just be a standard RT-11 BG program.

GET/SAVE are not required. These could be handled with a separate utility.

REENTER is perhaps useful, but at the cost of these incredible contortions not worth it.

E/B/D are so restricted that they are also not worth it.

Note that V5 has announced a SET KMON [NO]SWAP option.

DCL

"Walking on water wasn't built in a day" - Jack Kerouac

The DCL interpreter begins with a monster TECO macro, and then continues over many more pages to just get even more confusing.

(One advantage of the TECO macro is that TECO is required to put RT-11 together, this is one reason it is included in the standard kit).

I can't imagine that many more than three people have ever fully understood how the DCL interpreter really functions (one for V3, V4 and V5). This is a little bit like the story about the Principia Mathmatica - it was rumoured that only two people had read it in its entirety and it was not certain if this meant the authors).

DCL, and KMON, are just about the opposite of the Principia (I guess, since I am waiting till I understand DCL before moving onto Principia).

The name DCL has an interesting history:

CCL	Concise Command Language
DCLS	Digital Command Language Standard
DCL	Digital Command Language

Version three introduced DCLS with the following (first) sentence:

"RT-11 has an expanded set of monitor commands based on the DIGITAL command language standard".

Note how careful the language is ('based on').

Indeed, the 'S' has been dropped because there is no standard. RT-11, VAX and RSX are all different.

RSX for example does not display an 'Ambiguous' error message - it chooses one of the options instead. RT-11 is the only one of the three to support EXECUTE. VAX (and probably RSX) do not support factoring.

(RT-11 accepts P for PRINT, which is ambiguous with the PASCAL command. However, the reason for that was mostly so that we wouldn't stumble on to the fact there was a PASCAL command.)

Every language has a syntax. Usually, the syntax comes at the same time as the language. DCL syntax however had to be designed such that it would interface with all the existing CSI command formats.

Having had to reimplement DCL for a project, I can assure you that the syntax is a dog. Setting up the default devices, types, programs, prompts, continuations, plus versus comma, PIP/DUP/FILEX: you name it - it's horrid.

RT-11, cramped for space, sliding/swapping, has a tough job doing anything. DCL in this environment is one of the seven wonders of RT-11.

The DCL interpreter is a token-driven/table-driven interpreter.

The byte tokens include ifs and gotos, but no explicit counting mechanisms.

The tables describe the permissable command options, and the parameters for the options. The big TECO macro is used to pack the ascii information in the OPTION tables.

DCL internals could occupy more than a day. But, we would need one of the three people I mentioned before.

(However, the worst command to do, turns out to be the LIBRARY command - it has so many variants that I still haven't got it right).

USR

The USR is 2k words long. That figure was defined by RT-11 V1. Some programs allocate a 2kw space to overlay the USR during execution. Therefore, the size of the USR cannot be changed, without changing all such programs. Since there are some 50000 RT-11 installations, this could mean modifying hundreds of thousands of programs. Therefore, the size of the USR cannot be altered.

The 2kw allocated to the USR was more or less full with V1 of RT-11.

V2, V3 and V4 have all added functionality to the USR. V5 will add more. How do you add functionality without increasing the size of the USR?

The answer is, you optimize the existing code to require less space. When you look at the USR code you must always remember that you are looking at code that has been squeezed a hundred times to find an extra word.

One report has it that it takes a programmer two days to find a free word in the USR. Thus, if RT-11 requires a new routine of 10 words, then 20 working days are required to find the space. At present, looking for more space in the USR is a bit like trying to get blood out of a stone.

Wishlist items that would require additional code in the USR are usually doomed to failure - since the space is just not there.

The USR code is very difficult to read because it has been optimized so often for space. It is unlikely that anyone understands it fully.

The USR has conditionals for SJ, FB and XM. It also has an incredible number of conditionals for all the different flavours of MRRT - which makes reading quite painful.

Working on the USR could not be described as fun, and that perhaps explains the absence of humorous quotations in the USR sources.

When the RT-11 group are developing new code for the USR they need additional space for debugging. Thus, during the development cycle, the GETNAM routine is temporarily moved into RMON to gain that space.

During the rest of the USR discussion I will point out places that they have saved code. Some of these techniques have already been discussed in the KMON section.

USR BUFFER - 0.5kw

The USR buffer is located at the beginning of the USR. The USR buffer is used for two purposes:

Data Directory segment buffer, KMON i/o buffer
Code Once-only USR code

buffer data

The main reason for the USR buffer is to perform directory operations. The size of the buffer, 512 words, corresponds with the 2-block size of a directory segment.

This buffer is also used for KMON/KMOVLY purposes. For example, SET reads the first two blocks of a handler into this buffer to perform SET operations. Thus, the reason that SET reads only two blocks in is explained.

The USR directory routines keep track of which directory segment of which volume is present in the buffer. That is, the buffer acts as a cache. This sometimes saves rereading the directory if the segment is already in the buffer. This information is stored in the following RMON offsets:

```
CHKEY: .byte   device-slot,unit
BLKEY: .word   directory-segment
```

Since a segment number is always non-zero, the value zero in BLKEY is used to signify that no segment is in the buffer.

Note, that RT-11 does not have any mechanism for knowing when you physically change a volume. The worst case for this is a floppy with a single active directory segment. If you change such a volume, and the directory segment is in the USR buffer, RT-11 may report File Not Found, etc., since it is only looking at the segment in the buffer.

Previous releases of RT-11 would even go so far as to write out that segment to the new volume.

If you have this kind of problem, the solution is to purge the cache. You could do this by clearing BLKEY (with the USR appropriately .LOCKED), however, this might not work in all RT-11 environments. A better way is to look for a known file on the system device. Currently, to purge the cache I do a lookup to SY:SWAP.SYS.

buffer code

Since RT-11 is never quite sure whats in the USR buffer, it always rereads the USR before executing the code in the buffer. Thus, SET USR NOSWAP does not mean that the USR will not occasionally be reread from disk. This is particularly true for the SJ monitor.

The first routines in the USR buffer are used to relocate the USR code. This saves space since PIC code would be longer. The relocation tables are not easy to follow - but who needs to follow them?

The following routines are in the buffer:

```
SJ      FATAL  CDFN  SRESET  HRESET
SJ/FB   QSET
```

Note, that the USR is handled differently under XM. It need never be reread and may exceed 2kw.

The SJ FATAL routine prints monitor fatal error messages. It includes a SYSGEN option PANI$C. I have not seen this conditional described anywhere, and have never tried it. But, I would assume, from reading the code, that it would add a print out of the all registers to fatal error messages. It does not seem to have an equivalent under the FB monitor.

USRST: - USR Start

After the buffer comes the USR dispatcher. The first check is for USR recursion under the SJ monitor. I have the feeling that the FB monitor does not check for recursion at present.

The first good reason that the USR is not recursive comes next: Some parameters to the USR call are stored in the locations ARGM1, ARGM2, ARGM3. USREMT notes whether this is an old (V1) or new (V2) style EMT.

ARGM1 usually points to the filename for the request. Note, that this address is sometimes odd. That is, RT-11 CUSP's set bit 0 of this parameter to indicate that they wish to examine or modify the directory entry for the file. This is how [NO]SETDATE and [NO]PROTECT are implemented. In this case the fourth word of the EMT block will point at the routine to be called:

area:	.byte	chan,subcode	decoded
	.word	filnam-address + 1	R0
	.word	size or sequence	ARGM1
	.word	sequence	ARGM2
	.word	extra-word routine address	ARGM3

A spin-off of this is that RT-11 will not properly detect odd-address errors in the FILNAM block parameters. Instead it will handle these by calling an unspecified extra-word routine (which will probably be null - and thus go to the location zero exit code).

Note that this functionality could (and probably will) change in V5. If you use undocumented monitor features, you must always be prepared to change it when a new release comes out. You should only use such functionality when it is absolutely required.

Further, no guarantee can be made for the information detailed here. Before using such features you must be competent to understand just what is going on. I have not used some of these techniques with V4, and they may have changed in detail.

Indeed, the main reason for understanding this functionality is so that you can debug better, since sometimes debugging a program means debugging the monitor.

I would even more strongly advise against using this knowledge to modify the monitor. I put my last patch into RT-11 in V2B. Since then I have been able to workaround any monitor problems within my programs, or sometimes with special purpose handlers. Modifying the monitor is a thankless task and you always get burnt when a new release arrives. Always.

DIRECTORY ROUTINES

The common directory routines follow:

LOOKUP RENAME DELETE ENTER CLOSE

I would like to say "these are straightforward" and continue with other topics. But, nothing in the USR is straightforward - however, the operations they perform are fairly clear to us all.

Until fairly recently (in one of the V4 patchs), RT-11 had no checks in the USR to see if a directory was valid. This meant that looking-up an uninitialized volume (usually a floppy) would happily hang or crash the system.

Note, that if the system crashs while RT-11 is performing a directory segment expand/split, you could have a time bomb in the directory. The last thing RT updates is the pointer to the next free segment. Thus, after the crash, RT may not know about the new segment. Thus, the next time expansion/split occurs it will reuse a segment for a second time. This makes interesting reading of directory dumps.

SPESHL

Of particular interest is the way the USR performs directory operations for special directory devices. It does not, for example, use the .SPFUN defined for this function for MT:. Instead, it uses a special version of the .READ request. This format will also work from user programs - its only disadvantage is that it is not reentrant.

It builds the following EMT block:

```
area:   .byte   chan,read-subcode
        .word   sequence (zero for V1)   ;blk
        .word   filnam address           ;buff
        .word   size                     ;wcnt
        .byte   377,func                 ;function
        .word   0                        ;completion (wait)
```

The USR uses .READW - however, .READC works quite well also.

The function codes are positive. They are:

I seem to set the BATBIT in @CONTXT to disable address checking when I use these calls. I am, in retrospect, not sure why.

This technique is non-reentrant because the handlers return status via fixed RMON locations. They are:

SPSIZE	The size of the file
SPUSR	Error status (one word) (Standard RMON offset)

The routine to get the address of SPSIZE is:

```
        sysptr=54
        $mtps=362
gspsiz:
        mov     sysptr,r0
        add     #$mtps+2,r0
        movb    -2(r0),r1
        asl     r1
        add     r1,r0
10$:    cmp     (r0)+,#207
        bne     10$
        return                  ; r0 -> spsize
```

This is a dreadful technique - but the only one available at present.

FETCH, DSTATUS

dstatus

DSTATUS is a prerequisite for FETCH, since FETCH needs the information given by DSTATUS to perform.

DSTATUS has been reimplemented a few times. During V3 it used to reread the first two blocks of the handler into the USR buffer to get the required data. This was horrid, since many programs (such as PIP) used DSTATUS constantly to check whether a device was MT: or not MT:.

V4 stores all the data required for DSTATUS in the monitor tables. Indeed, this is one routine that could quite usefully move out of the USR into RMON. Thus making access to it faster, and providing extra space in the USR.

An example of the incomprehensible nature of large programs. DSTAT and FETCH used to have a common routine, RDDEV, to read the first two blocks of a handler into the buffer. This was no longer required in V4 for DSTAT. However, the code in FETCH did not change. It appears that this routine is now redundant - and occupies 13 words of precious USR space.

FETCH

FETCH is fairly easy. FETCH gets the following information from the monitor:

ARGM1 Address to load the handler
DVREC Disk block of handler (block 1 of handler file)
HSIZE Handler size

FETCH checks the SYSGEN options - in fact, this is redundant since INSTALL has already performed the check.

Having read the handler in, FETCH points to the last word of the handler, and pushs the following pointers in, if required:

erl$g=1
mmg$t=2
tim$it=4

1. fork routine
2. inten routine
3. timeout routine
4. error logging
5. memory management
   putwrd, putbyt, getbyt, mpphy, reloc

CSI

CSI is just about the only acronym that means the same thing on all PDP-11 operating systems: Command String Interpreter.

It is also one routine that really does not need to be in the USR. There is nothing in the CSI code that cannot be done by programs for themselves.

However, encapsulating this functionality in the monitor has spinoffs. For example, the RSTS RT-11 emulator decodes extended filename syntax in its CSI. I try to use the CSI for directory operations whenever possible.

command files

In V3, the CSI was extended to handle command file input and .GTLIN. (This needed a lot of USR space). Standard RT-11 supports command file input only via CSI calls (including GETLIN). Specifically, this means that you can't access command files via .TTYIN.

I got around this problem with a utility handler. The handler uses the E16LST to intercept TTYIN's. It then checks to see if a command file is active (and a few other tests). If a command file is active, it uses GTLIN to fill an internal buffer, and feeds these characters back to TTYIN calls. A fragment of the handler (which also performs other functions) is appended.

errors

No matter what the setting of .SCCA is, any ^C in a command file is final. This applies to most CSI errors, which also ignore SET ERROR NONE directives. CSI errors do not set the USERRB either. Why?

USR space restrictions always apply as an answer - everything costs space. The reason that ^C is final is simple: KMON stops storing command file data when it finds ^C - thus, after ^C it must return to KMON to pick up more input from the command file.

wildcards

.CSISPC will accept names with wildcard characters. These are represented in the RAD50 code with the following:

%	34
$	35

RMON

"Things are more like they are now than they ever were before."
Dwight D. Eisenhower

This quote introduces V4 RMONFB - and signifies that with V4, RMONFB had more or less become stable, but more, finally performed mostly as the system documentation purported.

The original quotation to the FB monitor after the title:

"The age ... of sophistors, economists and calculators, has succeeded."
Burke

The quote is apt. And RMONFB is the crown of RT-11. (I am also a fan of RMONSJ, though I find few others who like it).

The original FB monitor was written by Anton Chernoff for V2, and he returned to RT-11 for V4. Most of the quotations are also his work.

The FB monitor was written to be upwards compatible with SJ. There is almost no common code. And indeed, the common code that exists has usually moved from FB to SJ (for the timer etc.). At 4220 words, RMONFB is over twice the size of the 1996 word RMONSJ. System Jobs and XM make it somewhat larger.

RMONSJ is the minimal size monitor and suffers space constraints. RMONFB, while still forced to remain reasonable, is allowed to use all the space it needs to do the job properly.

Reading the V3 RMONFB sources was complicated by the XM and MTT conditionals. In V4 most of this code was removed to the separate modules XMSUBS and MTT. Both these modules should be considered a part of RMONFB.

$RMON

RMON begins, for both SJ and FB, with the RMON fixed-offset data base. Most of these are well described in the documentation. Lower case entries below are bytes.

	$RMON	JMP $INTEN is the usual contents of this location. The RSTS emulator has a zero in this location - which is used to signal that its RSTS. Other emulators also change the code slightly here for the same purpose.
	$CSW	The channel status tables. There are 17. channels here, not 16. However, the SJ monitor uses parts of channel 17 for its impure data. Channel 17 is used internally by RT-11.
	BLKEY	Discussed in the KMON section
	CHKEY	As above.
	$DATE	The RT-11 date.
	DFLG	Set to stop the monitor aborting in the middle of a directory modify (particularly an expand/split).
	$USRLC	Where the USR buffer is at present.
	QCOMP	Points to the handler i/o completion routine
	SPUSR	As discussed in SPESHL operations in the USR section
	SYUNIT	SY: device unit. Note, its in the high byte of this word.
	sysver	RT-11 Version. I don't know if anyone uses this.
	sysupd	RT-11 update.
	CONFIG	Well described.
	SCROLL	Points to GT database.
	TTKS	Points to physical TKS
	TTKB	
	TTPS	
	TTPB	
	MAXBLK	Maximum file size.
	E16LST	See below.
FB	CNTXT	Points to current jobs impure area.
FB	JOBNUM	Current job number
SJ	$TIME	Current time of day (2 words)
	SYNCH	.SYNCH entry pointer
	LOWMAP	Map of system protected words in low memory
	USRLOC	Another way of finding the USR.
	GTVECT	VT11 vector.
	ERRCNT	Error count return from programs. Currently unused.
	$MTPS	br putpsw
	$MFPS	br getpsw
	SYINDX	Slot number of system device
	STATWD	DCL and command file flags.
	CONFG2	Additional configuration information.
	SYSGEN	SYSGEN options included.
	USRARE	Size of USR in bytes. Always 2kw except for XM.
	errlev	current setting of set error
	ifmxns	Indirect File MaXimum NeSting level.
	EMTRTN	Address of emt return path - $rmon for BATCH E16LST routines.
	FORK	Address of $FORK - $RMON
	PNPTR	Address of $PNAME table - $RMON
	MONAME.	RAD50 /RT11FB/
	HSUFFX	Handler rad50 suffix (.rad50 / X/)
	DECNET	Reserved for the mysteries of DECnet.
	EXTIND	Low byte returns error status (a la USERRB) High byte 100 => KMON running IND; 200 => IND running KMON

IMPURE AREA

"Merely corroborative detail, intended to give artistic verisimilitude to an otherwise bald and unconvincing narrative."
W.S. Gilbert, The Mikado

Three pages are required to describe the impure area of a job. In fact, as far as job impure areas go, thats not very large - I know of one that requires 20.

A job can have at most two job processes running under the FB monitor. The root process a completion routine. The impure area allocates space to save root variables while a completion routine is running.

RT-11 requires per-job impure space to save data during context switchs. The FB monitor uses the jobs stack for this purpose. However, XM has problems finding the job's stack at the best of times, and thus allocates space in the impure area for context switchs.

RT-11 requires job variables to record such things as the address of the TRPSET routine, or the current state of the job.

RT-11 uses a bitmap to protect locations in the vector area. Under the SJ monitor the RMON fixed-offset BITMAP is used for this purpose. FB permits each job to .PROTECT vectors; thus each job requires a bitmap, and KMON must check the inclusive OR of all such bitmaps when loading a program.

All this data is collected together in the impure area for the job.

JOB NUMBERS

RT-11 can always find the current jobs impure area by looking at the CONTXT fixed-offset. The RMON table $IMPUR is a list of the start-addresses of all the impure areas. A job number (always a multiple of two) is an index into this table.

It is worth noting that the original FB monitor only had to deal with two jobs. However, the author laid all the groundwork for multiple jobs. Indeed, when it came time to do System Jobs, most of the essential code was already in RMON, and only a few parameters had to be changed. (there was substantial work in SRUN and in doing the new MQ: handler).

The RT-11 scheduler can handle up to 128 jobs - since it has a byte to store the even job number. However, in certain structures, the job number must be fitted into a byte with a channel number. This leaves four bits - enough for 16 jobs.

ERRORS AND OTHER FEATURES

"As far as we know, our computer has never had an undetected error."
Conrad H. Weishart, Union Carbide Corporation

The RMON code begins by handling errors. One of the major tasks of a monitor is to correctly detect errors, and dispatch them to user routines where possible.

Errors provide many of the best quotations:

"The fault, dear Brutus, is not in our stars, but in ourselves"
Shakespeare, Julius Caesar

"And oftentimes the excusing of a fault doth make the fault worse by the excuse"
Shakespeare, John IV

FPPERR:
FPU errors are processed in two stages. They have to be.

The floating point processor is asynchronous with the CPU. This means that it can detect an error after control has returned to the CPU. When the FPU finds an error it issues an interrupt. So what?

Here is what. FPU interrupts run at priority level 8 which is interesting since priority level 7 is supposed to the highest around. FPU interrupts can interrupt anything, which makes them hard to handle, since they could interrupt crucial code sequences that can't be broken.

Thus, the first stage of FPU error processing sets a flag, and if the CPU is at PR7, simply dissappears. It reasons, since we are at PR7, we must be in an interrupt sequence, therefore, our flag will be seen during the RMON exit path.

What this works out at is: If you have code that goes to PR7 with out using $INTEN, you could risk delaying or in the very worst case, losing FPU errors.

FPPERR is the second stage which passes the error to the job.

POWER FAIL, SYSTEM HALT

"Extreme remedies are very appropriate for extreme diseases."
Hippocrates, Aphorisms

At system halt the system is dead - thus:

"The death of God left the Angels in a strange position."
Barthelme, On Angels

However, all is not lost - there is still enough to debug on. RMONFB is careful not to modify registers during the error path (they suffer more monitor crashs during development than most users will ever see). In some cases the PC will have been popped off the stack into R4. Sometimes the stack is replaced with USERSP (location 40).

ERRCOM
This routine prints fatal error messages. One error message you should watch out for is 'Directory unsafe'; this means that the job died in the middle of a directory-modify operation and may no longer be consistant.

If you want to see a nice piece of subtle code, look at the routine that prints the piece address:

```
; print the number in r4
        mov     #30,r0
        sec
3$:     rol     r4
        rolb    r0
        .ttyout
        mov     #206,r0
4$:     asl     r4
        beq     5$
        rolb    r0
        bcs     4$
        br      3$
5$:
```

EMT DISPATCHER

"What's it going to be then, eh?"
A. Burgess - Clockwork Orange

One of THE hacks of RT-11 is that the TTCNFG word is located just before the EMT dispatcher. Thus, to locate this word you take the EMT vector and push down to TTCNFG. Rumour has it that TECO wanted it this way. Rumour has it that a substantial part of RT-11 is the way it is, because TECO wanted it that way.

The EMT dispatcher is a short piece of code that must have many weeks of work invested in it. The dispatcher must:

1. Work out what its going to be
2. Work out if its V1, 374, 375, 376 or 377
3. If its old, work out if its E16
4. Check some addresses
5. Not check SOME addresses
6. Setup channel pointers
7. Preset some registers
8. Call the routine

V1 EMT's have all the parameters on the stack. V2 uses R0 or a parameter block. The planned desupport for V1 requests did not take place because the additional overhead to handle both forms seemed bearable. Anyway, not all requests were converted to V2. CSI requests still exist only in V1 form.

RT-11/FB may take a couple of hundred instructions to dispatch an EMT - this is not real fast. In some cases it makes sense to bypass the monitor. For example, I have an application that serves upto 238 channels - all of which require close/purge fairly frequently. Rather than call the monitor to .PURGE a channel (or check to see if its open with .WAIT) I just find the CSW and clear it. Ten instructions instead of two hundred.

Other EMT's that I do by hand include .SRESET (as mentioned earlier) and DSTATUS (so that I don't have to wait for the USR). However, I do this only in critical situations.

V1 & V2

V1 program requests use three bits of the EMT to encode the channel. This limits them to 16 channels, and makes it difficult to write code that changes the channel number dynamically. V2 program requests use R0 or R0 as a pointer to a parameter block. We will use the .READW request as an example:

On entry to the dispatcher:

	v1		v2
oldpc-2	EMT+read+chan	r0=area	.byte chan,read
r0	block number	area+2	block number
n(sp)	buffer	area+4	buffer
n+2(sp)	wordcount	area+6	wordcount
n+4(sp)	completion	area+10	completion

RT-11 normalizes the request, and exits the dispatcher as follows:

(sp)	block number
r0	block number
r1 ->	buffer (n(sp) for v1, area+4 for v2)
r2	0 for v1, 100000 for v2
	USR sub-dispatcher uses r2
r3 ->	CSW area (if any)
r4	unrefined
r5 ->	impure area

SUB-DISPATCHERS

For many requests, the EMT dispatcher is not the end of the line. There are a number of sub-dispatchers:

EMT16

This dispatchs V1 requests that do not have a channel number. These are:

ttyin	ttyout	dstat	fetch	csigen	csispc	lock	unlock
exit	print	sreset	qset	settop	rctrlo	astx	hreset

Note that ASTX is used only in XM - it acts as an exit under SJ/FB.

BATCH uses the EMT16 dispatch list to get control of the following:

ttyin ttyout exit print

Your programs can also do the same. You locate the start of the list with the E16LST offset in the monitor tables. You then save the contents of the dispatch entry you want to modify in a .DEVICE list to make sure it gets put back when your program crashs. You locate the start address of the routine by adding E16LST to the value. And you replace the value with an offset relative to E16LST to your own routine.

Setting up calls via the E16LST is certain to provide hours of frustrating fun the first time you do it.

Your routine can either do the whole job and return to monitor. Or it can decide not to do it, and pass control to the standard RMON routine. If your program does perform the task, it must use the routine specified by the EMTRTN fixed-offset to return to the monitor.

USR

USR routines go thru this sub-dispatcher. This includes some routines that have just passed thru the EMT16 sub-dispatcher.

dstat rename lookup enter close2 fetch delete qset2

Note that CLOSE only comes here if a directory needs to be modified, or the channel was opened on a special-directory device. Notice, that purges to such devices do not come here.

CSI

Part of the FB CSI code must be in RMON. This part collects the input line for CSI requests that specify terminal input. If this were left to the USR, then the USR could be blocked/locked for hours while you came back from lunch.

KMON

While not truly a sub-dispatcher, this is a related topic, and there is space left on this page for the topic.

Some requests are eventually routed to KMON for final processing. These apply to BG programs only.

abort EXIT CHAIN

Since FG and System Jobs can also EXIT, a lot of the code for this request must be in RMON. And since it's there, BG uses it as well.

REQUEST PROCESSORS

The routines that process program requests are generally straightforward (yes, we can say that in RMONFB). The software environment is so well defined by the time that they get control, that they have little choice in what they do. Further, at this level, the world has been reduced to monitor tables and pointers - so, there is also little choice in who they do it to.

Here are one or two simple examples:

S$CCA

This routine processes .SCCA requests. On entry the address of the callers SCCA flag-word is on the top of the stack (see EMT dispatcher above). For FB systems, this value is simply moved into the impure area I.SCCA offset. For XM systems, the address must be converted to a PAR1+displacement two word address (unless it is zero).

Note that the name SCCA probably originally meant Set Control C Ast. However, we ended up with a flag-word rather than an AST.

S$ETOP

Life is a bit more complicated here. Having got the jobnumber (JOBNUM) the request is immediately rerouted to XMSUBS if its XM. The routine then checks to make sure the requested high address is in range. SETTOP then checks to see if its inside the current limits. If its under the jobs low limit, the requested value is changed to the low limit. If its under the high limit, it is accepted, otherwise it's changed to the high limit.

For BG programs, it then no so rashly assumes that we will overwrite the USR (since 99% of SETTOPs request all available memory) and calls ENQUSR to get control of the USR. If the USR is currently locked, or in use, by another job, then it could take some time to return. The BG job will be blocked waiting for the USR.

When control returns from ENQUSR, SETTOP has control of the USR and sets it in NOSWAP mode. This is perhaps not true - but since SETTOP has control of the USR, that is irrelevant.

SETTOP checks to see if the new BG high-limit will overwrite KMON (if it is still in memory). It it does, KMLOC is cleared, which signals that KMON is no longer in memory.

SETTOP checks to see if it really does need to swap the USR. If it doesn't, it simply exits, calling DEQUSR to make it accessable by other jobs. Otherwise, it now checks to see whether it is permitted to set the USR SWAP (since the terminal SET command can prohibit this). If it may swap the USR, then it sets the SWAP mode and exits (releasing the USR with DEQUSR). Otherwise, it takes the USR load address as the new value for SETTOP and loops back. This loop must succeed, since it is specifying an address lower than the USR.

In a program that had more room a different algorithm would have been used. In an RT-11 monitor, the algorithm that requires least space is used. Even if it requires redundant locking/waiting on the USR.

RESIDENT TELETYPE HANDLER

This is the part I dread. If you have no really good reason to get into the terminal handler, then don't.

The terminal handler is where the system interfaces with human beings. And, despite the fact that human beings are intelligent, or perhaps because of it, they are the most difficult peripheral to interface.

Some 26 pages of the monitor sources are occupied with this task, with hundreds of labels and conditionals. Probably the most complex part of adding System Job support was adding the CTRL/X functionality - since that meant figuring out what the terminal software was doing.

There is no way to describe this software wholly. Usually if you have a question with this code, it will be a quite specific question, and it will not be necessary to learn the whole routine.

Terminal interrupts are time-consuming. On an LSI-11/2 with a single interrupt priority, a terminal interrupt will block all other interrupts. This can cause real-time processes to time out, and it is often necessary to turn the terminal off during data acquisition. Otherwise, a random carriage return at the console could bring an experiment that had run quite happily for a couple of hours to its knees. Since disabling the terminal is per se dirty, the best way to do it is quick and dirty - via the TTKS/TTPS offsets.

You should however wait until RT-11 has completed any current terminal output before turning off the interrupts. You do this by waiting for bit 6 to go low - not bit 7. This is because bit 7 can set in the middle of an instruction.

One little hobby I have is looking for bugs in the monitor. And, I will set you the exercise of working out what the bug is in TTOPT2. This routine can be called by either TTYOUT/PRINT at PR0 to put a character in the output ring-buffer, or during input interrupts to put a character in the output ring-buffer.

high-speed input ring buffer

This routine is perhaps a more dignified way to get around the LSI-11/2 problem above. The high-speed ring-buffer stores characters in a separate ring-buffer and supplies these to the standard input routine at fork level.

READ & WRITE

"I'm quite illiterate, but I like to read a lot"
Salinger, Catcher in the Rye

"Their manner of writing is very peculiar, being neither from the left to the right, like the Europeans; nor from the right to the left, like the Arabians; from up to down, like the Chinese; nor from down to up, like the Cascagians."
Swift, Gullivers Travels

"'So it does!' said Pooh, 'it goes in!'
'So it does!' said Piglet, 'and it comes out!'"
Milne, Winnie the Pooh

I/O is the main reason operating systems were ever invented. RT-11 is very fast at this game, and the I/O routines are fairly simple.

Note, that the SJ monitor is still faster than FB. I had a problem with floppy benchmarks in BASIC one time. FB seemed to take six times as long as SJ. The reason was that FB was just slow enough to miss the sector skew on the floppies, and had to wait a full rotation for the sector to return. The main reason was BASIC.

The personal code for READ occupies only four instructions, two of which are NOP'sl. For the rest, it provides common code for both WRITE, SDAT & SPFUN.

All these routines use the common code TSWCNT to do most of the checking. WRITE must additionally set the C.USED entry in the CSW area (some cusps like KED also play with this variable).

After checking the I/O request, it is sent to the queue manager, to be placed on the queue of the requested device.

completion i/o problem
With V3 RT-11 started checking for eof/hderr both before and after a request was posted to the queue manager. This presents a special problem for completion I/O. Given that a .READC returns with an error, it is impossible to work out if that error was registered before the request was sent to the queue manager, or after it was sent to queue manager. The difference is very different. If the error was registered before the queue manager call, then no completion routine will have been scheduled. If it was registered after the queue manager call, then a completion routine will have been scheduled. In daisy-chained completion i/o this can create havoc.

My solution until now has been to clear all errors in the CSW before the request (this will only work if you only have a single I/O request active at a time). Then any eof/hderr errors must have occurred after the queue manager call, and a completion routine will be delivered. Note that system HERR's (error code is negative) are always registered before the queue-manager call.

QUEUE MANAGER
The queue manager takes an I/O request and converts it into an I/O queue element, which it places on the handlers queue. On entry to QMANGR, most of the parameters are in the registers:

r0 =	block number (from the emt dispatcher)
r1 =	unit number (in high byte)
r2 ->	fourth word of handler
r3 ->	CSW area
r4 =	wcnt (negative => write)
r5 ->	buffer, word-count

R1, R2 and R4 show what READ/WRITE and TSWCNT have produced since the EMT was dispatched.

QMANGR saves some registers and looks for a queue element for the request. The job goes in to wait-state if none is available. Which would be unfortunate if the job was in a completion routine.

It then fills in the I/O queue element.

q.link	clears this
q.csw	r3 goes here
q.blkn	r0 goes here
q.func	clears this at first
q.unit	high-byte of r1 goes here
q.jnum	job-number * 8 (overlays unit-number)
q.buff	(r5)+ goes here (relocated for XM)
q.wcnt	r4, now on stack, popped here (r5)+ popped past request wcnt
q.comp	(r5)+ goes here. If the low-byte is 377, then its really a SPFUN, and the high-
q.par	this is computed by XM in relocating q.buff

QMANGR then wants to put the queue element in the handler queue. Two cases emerge:

1. The handler is not busy. QMANGR sets it up as the first element and calls the handler. Finis.

2. The handler is busy. QMANGR hunts down the queue to find a place for the element. It searchs until it finds a request with a lower job number, or the end of the queue.

Note that in case two, RMONFB does not update the LQE entry of the handler.

This scheduling takes place in system state with the handler in hold-state. We will discuss both these states later.

QUEUE ABORT

Rather than discuss queue completion, I will discuss queue aborts. Aborting an I/O request is not a completely deterministic process under RT-11 and explains the need for the handler HOLD state.

A job exits. The monitor wants to abort all pending I/O for the job, and finds that the job has an I/O request in progress in a handler. Thus it calls the handler at its abort entry point to abort the request. The handler abort routine cleans up and exits the handler via the DRFIN routine.

Sounds simple in theory and it usually works.

However, when IOABRT calls the handler abort routine, it does so in system state at PR0. The handler could, in the abort-routine time-window, interrupt and complete of its own accord. Thus, you could have two handler completions, one as a result of the final interrupt and one as a result of the abort entry point.

Thus the handler-hold state. When the handler is in hold-state, COMPLT ignores all handler completions, and simply sets the complete-in-hold-state. Thus, on the return to the monitor I/O abort routines, multiple handler completions have been reduced to a single complete-in-hold-state flag. The I/O abort routine checks this flag, and calls for completion for the handler if its set.

However, there is a little worm in this code. This comes from RT-11's history, and is a good illustration of what operating system developers can inherit.

In V1 of RT-11 handler aborts were not really required. A job exit just meant RESET and start over.

In V2, the FB monitor had to live with this heritage, and developed the HOLD technique to stay compatible. However, the crucial assumption in V2 was that by the time control returned to IOABRT, that all activity on the handler would be finished.

In V3 this assumption became false with the introduction of the .FORK request.

Now, going back to the example above, if now the handler completes of its own accord and schedules a FORK routine, then this fork routine will not be scheduled until after the IOABRT code (which is in system state) is finished. Then when the fork routine is finally scheduled it could try to complete the next i/o queue element for the handler.

This is not an undocumented feature of RT-11, it is a bug.

INTERRUPTS

"Life is made of interruptions"
Gilbert, Patience

"For sleep, health and wealth to be truly enjoyed, they must be interrupted."
Richter, Flower, Fruit and Thorn pieces

"Allow time and moderate delay; hast manages all things badly."
Statius

$INTEN

This is the famous 10% of the code that executes 90% of the time (not really - the idle loop fits this description better).

Speed is of paramount importance here. All devices are supposed to interrupt at PR7, and come here to lower priority. Obviously $INTEN's task to get back to them as soon as possible. The code here (for a Q-BUS machine), is as follows:

```
        jsr     r5,$inten
        .word   ^C<PRn>&PR7

$inten: mov     r4,-(sp)        ;;; save r4
        inc     (pc)+           ;;; bump level pointer
intlvl::.word   -1
        bgt     1$              ;;; branch if already switched stacks
        mov     sp,(pc)+        ;;; save users's stack pointer
tasksp::.word   0
        mov     (pc)+,sp        ;;; switch to system stack
rmonsp::.word   rmstak  ;**boot**
1$:     mov     r4,-(sp)        ;;; save r4 for the ps modify
        mfps    r4              ;;; get the ps
        bic     (r5)+,r4        ;;; down to handler priority
        mtps    r4              ;;; set the ps
        mov     (sp)+,r4        ;;; get r4 back again
        call    (r5)            ;;  call the handler
```

This routine, including the call, requires 21 memory references (note the easy way to compute instruction times is just by the number of memory references). I guess that's about 50 usecs on an 11/23 and 100 usecs on an 11/03.

R5 and R4 are saved on the stack at the time of the interrupt. In any case they belong to the job/system running that stack.

If this is the first interrupt, then INTLVL will increment from its initial value of -1 to 0, triggering a stack swap. Note, that the .ROM globals are largely irrelevant in the code.

The comment ;**boot** means that this variable has to be setup when the system bootstraps.

Why not use a MTPS PRn instead of the three step sequence above? Maybe something to do with the trace bit. Additionally, this code must occupy the same amount of space as the equivalent PS method:

```
        mov     #ps,r4
        bic     (r5)+,(r4)
```

And the reason this code is a word longer than it need be is because it has to the same size as the equivalent MTPS code. Sometimes, hidden in these mysterious little pieces of code, are undocumented engineering screw-ups. For example, the 11/40 is very sensitive about the way you address its PS word. When you change mysterious code in the monitor, or do it yourself instead of calling the monitor then you risk bumping into engineering screw-ups.

SCHEDULING

'Scheduling' is the interaction of a number of separate processes, rather than the task of a single procedure. We will go into this subject in detail.

Two of the basic concepts here are: (1) Preempting; (2) Retiring

Preempting
Preempting is an agressive force. A new task preempts or displaces the current running task in the machine. The new task is permitted to take over because it has a higher hardware or software priority than the task it displaces.

Retiring
Retiring is a passive force. It may lead to nothing more important than the idle loop. Retiring is the way a preempting task gives control back to a lower priority task. It may give control back to the task it initially preempted, or during its execution, the preempting task may have altered the system state such that on retirement the system gives control to a task different to the task that was preempted.

Interrupt
In a simple system structure, a preempting force always begins as a hardware interrupt. In any system, the majority of preempting forces are the results of interrupts (clocks, disks, terminals).

Now, if you want a system where a preempting force can originate in software, rather than as an interrupt, the easiest way to this is to use a psuedo-interrupt in the software. Thus, in our discussion we can consider all preemptive actions to be the result of a hardware interrupt or a software pseudo-interrupt.

In a steady-state system, if no interrupts occur, the system will stay in steady state. Turn off all the disks and clocks on your system to try this out!

A more important corollory is that the best way to focus on scheduling is to follow the possible paths that an interrupt can follow.

Moving up the ladder
Since the ladder is a priority system, a task can always be sure that no-one is executing higher on the ladder. Therefore, a task can move freely up the ladder. The restriction is that it must inform the monitor that it has moved up the ladder. Therefore, to move up the ladder, a task moves right up to PR7 with a psuedo-interrupt, and then moves back down the ladder, under monitor control, to the desired step. This is the pseudo-interrupt.

Moving down the ladder
A task must obey certain rules in moving down the ladder.

A task may never move below its hardware priority without monitor assistance.

For example, given a PR5 task is preempted by a PR6 task. Then if the PR6 task decides to move to PR4, you could have the situation where a PR4 task was preempting a PR5 task.

Therefore, interrupt service routines use the FORK mechanism to move down the ladder. The task is placed on the FORK queue and other interrupts are permitted to complete. When all the interrupts are done, tasks on the FORK queue are processed.

The other way for a task to move down the ladder is to pass its work onto some other task further down the ladder. Thus when an interrupt service routine requests a completion or synch routine, it is actually requesting someone lower down the ladder to take the work over. Thus calling for completion ends in the retirement of the interrupt and a change in system state.

SYSTEM STATES

In the previous scheduling section we dealt with operations that were mainly the results of 'actions'.

However, in the last paragraph, we dealt with the completion mechanism. Another way of looking at the completion mechanism, is by seeing it as a mechanism that transforms an 'action' into a 'state' ('completion' is used in a generic sense here, rather than just meaning handler completion).

For example, a handler completes an I/O transfer and calls the completion manager. The completion manager translates this act of completion into the alteration of the state of the job waiting for that completion. It unblocks the job.

The contrast is as follows. Above system state, in the interrupt world, priority originates and maps onto hardware priority. Below system state, in the job world, priority is determined by the state of the jobs as mapped into a number of system variables (I.STATE, I.BLOK).

The two worlds, of interrupts and jobs, are quite different. Interrupts have a limited repetoire of options available to them (no program requests), but life is also much faster in the interrupt world. Priorities change very quickly in this dynamic environment.

The job world moves much more slowly. Swapping between jobs requires a rather long context switch whereas swapping between interrupts requires less than 25 memory references. Only one job runs at a time - whereas up to four interrupts may be nested at any given time. Finally, the interrupt world is based on the machines hardware whereas the job world is based on the monitors transformation of the interrupt world as represented in its tables. Once again, it is the completion process that maps the interrupts actions into changes in those tables.

As you might imagine, the interrupt world, being based on hardware reality, is very similiar in all PDP-11 operating systems. There just are not many different ways to handle an interrupt (although I have seen some methods that would make you wonder about that).

Where operating systems differ is in the job world - which is essentially a creation of the monitor.

SYSTEM STATE

Before dealing with system states (note the 's') in detail, we have to recognise a third-world: System State (no 's').

System State is the middleperson in between the interrupt and job world. It has two main functions.

Firstly, it augments the hardware machine by simulating a couple of hardware devices the engineers didn't supply.

Secondly, it provides the monitor with a safe place that is neither in the interrupt world or in the job world. It uses this environment to change states of both the interrupt world (IOABRT) and jobs.

Interrupt world

$INTEN:

Process FPU	Delayed interrupt
Fork routines	(These are a cut above system state)
Process the clock	This is a psuedo-Fork routine
Abort jobs	Catch up with delayed aborts
Schedule jobs	Catch up with changes to system states
Context switch	Swap job context
Schedule completion	i.e. completion routines

$ENSYS:

Psuedo-interrupt	$ENSYS
Schedule I/O	Start a handler
Block job	

Job world

COMPLETION

Completion means changing the state of a jobs runnability. This is a table driven operation.

We will consider an example. A job issues a .TTYIN request for a character. None is available, so T$TIN calls $SYSWT to block the job with TTIWT$.

"Now the serpent was more subtil than any beast of the field." Genesis 3:1

$SYSWT is well described in the SSM (3-31!). What is not obvious is that when job becomes runnable again, it execute $SYSWT rather than the continuation of T$TIN. This is ensured by the call to $ENSYS (which also retains the original value of R4 for $SYSWT when it restarts).

The job is now blocked. A user, any user, types a character at the terminal, which causes TTINC3 to call UNBLOK to unblock the job. UNBLOK causes a scheduling pass which returns control to $SYSWT, which then runs the decision routine again. The paths are:

Retire	T$TIN, $SYSWT, $ENSYS, SWAPME, $RQSIG, ...
Restart	TTINC3, UNBLOK, $RQTSW, INTACT, EXUSER, EXSWAP, CNTXSW, $SYSWT, T$TIN, $SYSWT, T$TIN job

$SYSWT blocks the job.

TTINC3 is the interrupt, and UNBLOK is the completion component.

$RQTSW thru CNTXSW restart the job. If the character was not EOL, and T$TIN wanted EOL, then T$TIN and $SYSWT retire the job again. (Which means that everytime you type a character to the FG you cause a context switch, even if the FG program doesn't get control).

SYSTEM STATES

Now we finally come to deal with system states, we find we have already dealt with them.

The system ladder

You can picture the priority system of an operating system as a ladder. Tasks higher up the ladder have precedence over the tasks below them.

Further, you can picture this ladder leaned up against the wall of a multi-story building with a number of balconies. Tasks can retire to the balconies giving control to tasks on the ladder below them. However, by remaining on a balcony a task has the potential to enter the system at some later stage at a high priority. These balconies are represented in operating systems by queues.

The system must provide mechanisms for climbing up and down the ladder.

An interrupt is born

I tend to think of the total interrupt behaviour of a machine as Flak. However, this is rather well-behaved flak, that has to get landing permission.

In our model, an interrupt starts somewhere undefined in the sky. It goes into a flight loop until it gets permission to land. On top of our building is the Flak Flight controller. He requests from the interrupt the step on the ladder required for landing. Air traffic rules do not permit an interrupt to land lower than someone already on the ladder (This is hardware).

Communications between the flight controller and the ladder personnel are not very good. Thus, when ever permission is granted to land, the flight controller temporarily blocks all other landings until the ground crew on the ladder signal that the interrupt has safely landed (all interrupts occur at PR7 - $INTEN is the ground crew).

However, every rule has its exception, and there is one type of interrupt that can always land - Mr. F.P.U.. The ground crew get rid of him as quickly as possible.

the interrupt has landed

When a regular kind of interrupt lands, the first thing it does is report to the ground crew which step on the ladder it has landed on. The ground crew send this information to the flight controller to permit other interrupts to land higher up.

This takes time, so some interrupts who really only want to make a very short visit, don't bother to tell the ground crew. They just do their job and take off again. The flight controller can detect this and it causes no problems (except that it might have an affect on the special handling the ground crew give to Mr. F.P.U., if Mr. F.P.U. lands in the middle of one of these short visits.)

Once an interrupt has landed on a particular step of the (hardware) ladder it can start to do its work. If its work is going to take a long time it might to decide to move down the ladder. But, it can't do this by itself, since there could be someone on the step below it. It must ask permission of the ground crew to move down. In fact, the ground crew has a single level that interrupts can move down to (the fork queue - note VAX has four fork queues).

However, an interrupt can move freely up the ladder and back to its original position, since, by our rule above, it can be certain that no-one is on a step of the ladder above it. In practise an interrupt tends to make only trips to the top of the building (PR7) and back to its original step.

Eventually, perhaps after having being preempted a couple of times, an interrupt will either retire from the ladder by taking off, or call the ground crew to move down the ladder. If it does not, then the whole ladder will come to a stop. When an interrupt takes off, or retires, control generally returns to the next lower interrupt or task on the ladder.

Alices interrupt restuarant
When an interrupt wants to come down the ladder, it goes to Alices Interrupt Resaurant on the fifth floor. Since this is an American Cafe, it has to wait in line for service (fork queue). Note that it now loses all priority; it's just another interrupt waiting in line. In fact it's no longer an interrupt.

Ricks fast-food place
Actually, there is another place it can go to. Since RT-11 is a fast system, it also provides a fast-food place. However, the food is so terrible there that no-one actually eats it. They just come in to make a quick telephone call to the COMPLETION department to tell them that all their work's been done. In this case the COMPLETION department may give the interrupt another job to do (i.e. schedule the next queue element for that balcony).

after the interrupts over
Whenever there is an interrupt reported, the ground crew send someone to wait outside Alice's. This supervisor waits until the last interrupt has taken off, and checks to see if they left any mess lying around. The first thing they check is the fork queue. Each person on the fork queue gets a chance to run until there are no more.

Then our fifth floor supervisor checks to see if the clock has changed, whether Mr. F.P.U. left his visiting card, and finally the bereavment notices for jobs deceased are checked. All of these activities take place on the fifth floor. (The fifth floor is System State)

"It is not enough to be busy; so are the ants. The questions is, what are we busy about?" Thoreau

The last task of the supervisor is to return control to the guys on the ladder below. Now, interrupts, forks, clocks, Mr. F.P.U, and bereavements could all have changed the working roster. It could be that one of the people waiting on the balconies below can now move back out onto the ladder and start washing those windows again. Thus the supervisor checks to see if any of the people sunning themselves on the balcony should get back to work.

If one of them should, and they are highest up the ladder, the supervisor rearranges things (context switch) and calls them out to get going.

When one of these people under the fifth floor is done, they call up to the supervisor and ask permission to go back to the sun-shades and gin and tonic. The supervisor then looks for the next person to send out on to the ladder.

"To be idle is the ultimate purpose of the busy" Samuel Johnson

When everybody on the lower four floors is relaxing with nothing to do, the supervisor gets very busy with the Idle Loop. The supervisor frantically looks up and down the ladder for some good reason to get somebody out on the ladder. Since this is the case 90% of the time, you can imagine that the supervisor is kind of paranoid. Time is Lunch.

SCHEDULING PLAY

The following is a dramatic one-act play based on the IOABRT/FORK problem. There are two reasons I choose this scenario: (1) A problem sometimes shows the structure better; (2) I have this scenario in my head at present, so it's easier to put this together; it wasn't easy - even though it is simplified, the scenario still requires over 25 steps.

The scenario is as follows:

BG runs
BG .READ goes to Handler queue
BG read starts, enables interrupt

FG runs, BG waits
FG read goes to Handler queue

BG exits
IOABRT sets handler HOLD state
IOABRT to goes to Handler abort
BG read interrupts and goes to Fork queue
IOABRT request goes via drfin to COMPLT
COMPLT sets handler hold/completed state
IOABRT clears hold/complete,
  exeunt BG read request
  starts FG read request
FG read starts, enables interrupt
IOABRT exeunts

BG fork starts, exits request via drfin
COMPLT removes FG read request

FG read interrupts, no queue element

We will need:

BG	BG.read BG.rea2 BG.exit(doubled by BG)
FG	FG.read FG.rea2
IOABT	IO.abt
DR	

DR remains seated and displays only the state of the handler (run/hold/complete)

	JOB	SYSTEM	AST	INT	
kmon	jobwait	forkq	drlqe	intwait	seated
			drcqe		seated
			drbeg		
	job	ioabrt	complt		
			drabt	drast	
			drfin	drfin	system-halt
			$dr		

The action is:

step	actor	from	to	to	because
1.	FG	kmon	job	jobwait	FRUN FG, .twait FG
2.	BG	kmon	job		RUN BG
3.	BG.read	job	drcqe	intwait	BG .READ, startio, pause
4.	BG	job	jobwait		FG .twait done; context switch
4.	FG	jobwait	job		
6.	FG.read	job	drlqe		FG .READ, wait in lqe
7.	FG	job	jobwait		FG .twait, context switch
7.	BG	jobwait	job		
8.	BG.exit	job	IOABRT		BG .EXIT, abort i/o
9.	DR	$run	$hold		IOABRT
10.	IO.abt	system	drabt		IOABRT
11.	BG.read	intwait	drast	forkq	interrupt, preempts drabrt
12.	IO.abt	drabt	complt		fork queues behind ioabrt
13.	DR	$hold	$hold/complete		IO.abt/complt & hold
14.	IO.abt	complt	IOABRT		IO.abt done
15.	DR	$hold/c	$run		IOABRT
16.	BG.rea2	DRcqe	exeunt		CMPLT2
17.	FG.rea2	DRlqe	DRcqe	iostart/intwait	Scheduled
18.	IOABRT	system	exeunt		done
19.	BG.read	forkq	system	drfin complt	schedule fork
20.	FG.rea2	DRcqe	FG.done	exeunt	FG request completes
21.	FG.read	intwait	drast	fork complt	
		complt	system-halt		no queue element
					all actors remain standing

DSIR
DEPARTMENT OF SCIENTIFIC AND INDUSTRIAL RESEARCH

APPLIED MATHEMATICS DIVISION

P.O. Box 1335 Wellington New Zealand
Telephone (4) 727 855 Telex 3276 Research
7th Floor Rankine Brown Building Victoria University of Wellington

7 August, 1984

Ken Demers
Adaptive Automation
5 Science Park
New Haven, CT
06511
U. S. A.

Dear Ken:

I have often been annoyed by the fact that when a bad block occurs on a floppy disk during a read operation, the whole block is flagged as bad by the handler, even though in general it is only one sector out of 4 (or 2 for double density) that has caused the error. Further I consider it useful to be able to look at the bad sector anyway, since it is possible that only a small proportion of those 128 bytes (or 256 bytes for double density) are in error.

To this end I have modified the DY handler for RT-11 V5.0 to attempt to continue with data transfer process after an error has been detected, while still flagging the error in the normal way when the I/O request is returned. This means that all those good sectors within the 'bad' block are transferred correctly, and that the information read from the bad sector is also transferred, so that the user may determine how much of it really is bad. (At the moment a COPY/IGNORE command to copy a file containing a bad block in which it is the first sector that is faulty will repeat the block previous to the bad block when writing the output file).

The accompanying listing is of an input file for the Source Language Patch Program (SLP.SAV). There are three different patches identified:

Patch 001 is the standard TSX-Plus V4.1 file TSXDY.SLP for patching RT-11 V5.0 DY.MAC. This has been included for completeness, since TSX-Plus users must install this patch if any other patches are installed.

Patch 002 implements the mechanism whereby errors are reported if the retry count is exhausted, but are ignored internally by the handler in the sense that the process of data transfer is continued after an error has occurred.

Patch 003 is a further optional enhancement which implements the option

SET DY [NO]IGNORE

This causes the handler to not report bad blocks at all when they occur, although the retry process is performed as usual, and the information is transferred as usual, as provided by patch 002. This allows programs which terminate on detecting bad blocks to continue with whatever data has been provided by the handler. One side effect of using the SET DY IGNORE option is that some diskette drive hardware may behave differently, because there is no hardware initialise command executed after the error has been detected. Thus for example, a DSD-440 drive will flash the red LED for two minutes after the bad block has been detected. Patch 002 must be installed if patch 003 is included.

```
-125
        .DRSET  IGNORE, <BIS    (PC)+, @-(R4)>, O.IGN,  NO              ;003
-210
O.IGN:  MOV     (PC)+, R3                                               ;003
        TST     (PC)+                                                   ;003
        MOV     R3, O$IGN                                               ;003
        BR      O.GOOD                                                  ;003
-314,314
        ADD     #2, R0          ; INCREMENT BUFFER ONE WORD.            ;001
-384,384
        CMP     Q.BUFF-Q.BLKN(R4),#140000; CHECK PAR6 LOWER BOUNDARY.   ;001
-435
        MOV     DYCQE, R4               ; GET QUEUE ELEMENT.            ;002
        BIT     #HDERR$, @-(R4)         ; IF HARD ERROR BIT IS SET,     ;002
        BNE     DYABRT                  ;  WE MUST INITIALISE DEVICE.   ;002
-461
        BR      INTDSP                  ; NO ERROR, CONTINUE.           ;002
SETERR: MOV     R4, -(SP)               ; SAVE R4.                      ;002
        MOV     DYCQE, R4               ; GET QUEUE ELEMENT.            ;002
O$IGN:  BIS     #HDERR$, @-(R4)         ; SET HARD ERROR BIT.           ;002
        MOV     (SP)+, R4               ; RESTORE R4.                   ;002
-508
        DEC     DYTRY                   ; IF RETRY COUNT IS EXHAUSTED,  ;002
        BEQ     SETERR                  ;  CONTINUE AS IF NO ERROR.     ;002
-511
        JMP     DYINIT                  ; GO TO RETRY OPERATION.        ;002
-513,513
        BEQ     SETERR                  ; EXHAUSTED, SO CONTINUE        ;002
/
```

Of course no guarantees can be given with these patches, but they do seem to work on those bad blocks that I have tried. Good luck!

Regards,

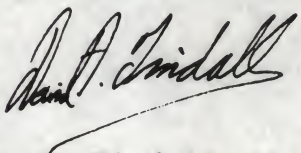
Roy Brownrigg

Dr. R. D. Brownrigg
Scientist - Computing Section.

I thought that the readers of the "Mini Tasker" might be interested in the enclosed program.

I have recently set up my RT-11 system to be used with a remote console terminal, with a dial-up connection over regular phone lines. Clearly anyone with a terminal could dial up, log in and wreak havoc with that system. The enclosed program gives a modest improvement in security with little effort: after running the program "SECURE" the console terminal cannot be used until the correct password is typed. Of course, one has to remember to run SECURE before hanging up the phone, but with this restriction it seems to work quite well. I have given it a modest degree of testing and it seems to be bug-free; however I invite readers to let me know if there are problems or improvements which could be made.

Yours sincerely,



DALHOUSIE UNIVERSITY
DEPARTMENT OF PHYSICS

Halifax, Nova Scotia, Canada, B3H 3J5
Departmental Telephone: (902) 424-2337
Telex: 01921863

D. A. Tindall
Assistant Professor

```
        .TITLE  SECURE
        .ENABL  LC
;
; a MACRO program designed to restrict the use of the RT-11 system console
; to password holders. It is particularly designed for applications where
; the console port is attached to a modem, to prevent unauthorised remote
; dial-up use of the system.
;
; Written by David A. Tindall (Physics Department, Dalhousie University,
; Halifax, Nova Scotia, Canada B3H 3J5) 1984-10-03 as a result of a
; suggestion from Philip Staal of D.R.E.A., Dartmouth N.S.
;
;
        .MCALL  .EXIT,.SCCA,.TTYIN
START:  .SCCA   #MAREA,#SCCA    ; Disable Control-C abort
        JSW=44                  ; Job Status Word
        PASLEN=PASEND-PASWRD    ; PASword LENgth
        BIS     #50000,@#JSW    ; Set lowercase and special mode bits
LOOP:   CLR     R5              ; R5 counts correct characters input
NEXT:   .TTYIN                  ; Get next character typed
        CMPB    R0,PASWRD(R5)   ; Does it match next character in password?
        BNE     LOOP            ; If not then start again
        INC     R5              ; Here if match found - move pointer
        CMP     R5,#PASLEN-1    ; End of password?
        BLE     NEXT            ; If not go back for another character
STOP:   .EXIT
MAREA:  .BLKW   5
SCCA:   .WORD   0
PASWRD: .ASCII  /Open Sesame/   ; Replace "Open Sesame" with desired password
PASEND: .END    START           ; Password can be any length
```

USER RESPONSES

Following are 'improved' versions of the macros submitted by John T. Davies III in the August minitasker. This version allows the use of any GP register as stack pointer for floating point operations and copies the floating point condition codes to the PSW so they may be used as conditions for branches. It now behaves more like the FIS on the 11/03. Many thanks to John for presenting this idea.

```
;
        .MACRO  FADD,X
        LDFPS   #7440
        LDF     (X)+,R0
        ADDF    (X)+,R0
        STF     R0,-(X)
        CFCC                    ; copy FP condition codes to PSW
        .ENDM
;
        .MACRO  FMUL,X
        LDFPS   #7440
        LDF     (X)+,R0
        MULF    (X)+,R0
        STF     R0,-(X)
        CFCC
        .ENDM
;
        .MACRO FSUB,X
        LDFPS   #7440
        LDF     (X)+,R0
        SUBF    (X)+,R0
        NEGF    R0
        STF     R0,-(X)
        CFCC
        .ENDM   FSUB
;
        .MACRO  FDIV,X
        LDFPS   #7440
        LDF     (X)+,R0
        LDF     (X)+,R1
        DIVF    R0,R1
        STF     R1,-(X)
        CFCC
        .ENDM
```

Lester R. Shields
Weirton Steel Corporation
Computer Process Control
Weirton, WV 26062

DECUS LIBRARY

revision
11-657

FTALK

Version: V1.0B, June 1984

Author: Timothy W. Coressel, Rockwell International, Golden, CO

Operating System: RT-11 V5.0

Source Language: MACRO-11

Memory Required: 3KW

Special Hardware Required: Two DL ports (i.e., DLVII-J)

FTALK is a software package for linking a development PDP-11 computer to a SBC 11/21 (Falcon) computer used in dedicated type applicatons. This program allows a user to download stand-alone programs from any mass storage device existing on a PDP-11 computer to the Falcon. It also allows one terminal to communicate to both the PDP-11 computer and the Falcon.

Changes and Improvements: FTALK is now able to download programs to the Falcon that consist of 2 blocks of more of executable code.

Documentation on magnetic media.

Media (Service Charge Code): Write-Up and Listing (DA), Floppy Diskette (KA), 600' Magtape (MA)

Format: RT-11

revision
11-720

STONE: A Program for Resolving Mossbauer Spectra

Version: July 1984

author: A.J. Stone, K.M. Parkin and M.D. Dyar

Submitted By: M. Darby Dyar, Massachusetts Institute of Technology, Cambridge, MA

Operating System: RT-11 V4.0, MINC V1.2

Source Language: FORTRAN IV

Memory Required: 40KB

Special Hardware Required: (Two) RX02 floppy disk ports or storage greater than 1600 blocks

This program is an overlaid, condensed version of the larger program MOSSPEC, which is in use worldwide. It fits a sum of Lorentzian or Lorentzian/Gaussian combined lines to a given Mossbauer spectrum by means of the Gauss non-linear regression procedure with a facility for constraining any set of parameters or linear combination of parameters. Results are output as a table of the fitted parameters, including the statistical values for standard deviation, x^2, and Misfit.

This program was developed to enable sophisticated Mossbauer curve-fitting to be done on a very small computer, such as would be present in a laboratory environment. Use of the detailed User's Manual is strongly recommended.

Note: Use of the accompanying manual is strongly recommended.

Changes and Improvements: Corrected bug in subroutine STOSTA.

Restrictions: Program requires approximately 900 blocks of free space for swapping on and off the disk; therefore it can only be run off RX02 (double density) floppies: one floppy for the operating system +STONE.SAV and one floppy for the few .TMP files and the empty blocks (contiguous). This program can also be run off any storage medium with a total of 1600 blocks.

Media (Service Charge Code): Manual (EA), RX02 Floppy Diskette (KA), 600' Magtape (MA)

Format: RT-11

new
11-SP-68

C Language System in RT-11 Format

Version: November 1983

Author: David Conroy, Robert Denny, Charles Forsyth, Clifford Geshke, and Martin Minow

Submitted By: N. A. Bourgeois, Jr.NAB Software Services, Inc., Albuquerque, NM

Operating System: RSTS/E V7.2/V8.0, RSX-11M V4.0, VMS V3.2, TSX-PLUS V2.2-V5.0, RT-11 V4.0/V5.0

Source Language: C, MACRO-11

Memory Required: 56KB

Special Hardware Required: Floating point operation requires FPU.

This tape contains the "C Language System, Second Master Release Version of November 1983", the same information as is contained on the tape for DECUS PART NO. 11-SP-18. The information is simply repacked from the DOS format of 11-SP-18 into an RT-11 compatible format. The files from each of the 14 DOS [UIC] accounts are contained in RT-11 Logical Disk files. The files from DOS [5,1] are in the RT-11 LD file 501.DSK. The logical disk files are all full with the largest being 3179 blocks and the smallest being 159. See the catalog listing of 11-SP-18 for further information.

Note: Hardcopy documentation is available. See DECUS part no. 11-SP-18 for ordering information.

Restrictions: DECUS C supports a subset of the current version of C. Minor problems may be encountered in converting from other dialects of C.

Documentation on magnetic media.

Media (Service Charge Code): 2400' Magtape (PC)

Format: RT-11

revision
11-424

CIT101: Routines to Drive the CIT101 and VT100 Terminals

Version: V2.2, March 1984

Author: Ralston W. Barnard, Sandia National Labs, Albuquerque, NM

Operating System: RT-11 V5.0

Source Language: FORTRAN IV, MACRO-11

Memory Required: 1328 (10) wds - max.

Special Hardware Required: VT100 terminal, CIT-101 terminal

CIT101 is a MACRO subroutine which is an extension of the DECUS Program Library Number 11-424, "VT". It is a collection of FORTRAN-callable routines for control of the VT100 terminal functions. Additions to this package include support for the enhanced screen-control features of the CIT101 terminal (including both the 10A and 11C ROM sets), and for the CIT101 and VT100/102 printer ports. Both of these extra features are individually available as conditional assemblies in the source code. The original submission has been recoded to reduce its size. The code has also been divided into "I and D" PSECTs.

The original demos provided with 11-424 are included. Also included are three routines which use the CIT101 library - for centering text, displaying text in two columns, and for putting out a string of numbers after a prompt. The file CICALL.TXT is a handy summary of all the calls.

Changes and Improvements: Enhancements to functionality; improved efficiency and size.

Documentation on magnetic media.

Media (Service Charge Code): Write-Up (AA), Floppy Diskette (KA), 600' Magtape (MA)

Format: RT-11

revision
11-444

Complete File Sort Utility

Version: V3, July 1984

Author: John M. Crowell, Crow4ell, Ltd., Los Alamos, NM

Operating System: RT-11 V4.0, 5.0, 5.1

Source Language: MACRO-11

Memory Required: 16KW

Special Hardware Required: EIS

RTSORT is a substantial revision of DECUS Library Program 11-444, originally by Bob Schilmoeller and Paul Styrvoky of St. Johns's University, Collegville, MN. The program performs a multiple key sort of a data file in either alphabetical or ASCII order. The sort is accomplished via a Tag Array built with the specified sort fields and block and record addresses. A Shell Sort puts the Tag Array in ascending or descending order. The sorted data is written to a file, and, optionally, printed on the termial.

A maximum of 16 sort fields is allowed. Maximum record length is 2046 bytes. Records must be separated by a /CR/LF.

In the preparation of this version, no changes in the sorting procedure were made. Revisions consist of the following:

1. Replacement of redundant code with subroutines.
2. Improves decimal/ASCII conversion.
3. Runtime memory allocation.

The results of these revisions are:

1. Up to 30% increase in maximum number of sorted records.
2. Size reduction of SAV image from 60 blocks to 6 blocks.

A maximum of 16 sort fields is allowed. Maximum record length is 2046 bytes. Records must be separated by a /CR/LF.

Documentation on magnetic media.

Media (Service Charge Code): Floppy Diskette (KA),
600' Magtape (MA)

Format: RT-11

revision
11-731

Kermit-11

Version: V2.20, August 1984

Author: Brian Nelson

Submitted By: Rebecca Dent, Change Software Inc., Toledo, OH

Operating System: RSTS/E V7.2 or later, RSX-11M V4.0 or later, RSX-11M-PLUS V2.1, RT-11 V4.0 or later

Source Language: MACRO-11

Memory Required: 20KW

Kermit is a protocal originally developed at Columbia University which has been used to implement error free packet file transfer and communications between computer systems, both mainframe to mainframe and micro to mainframe.

This Kermit-11 was developed by the author for RSTS/E, RSX-11M-PLUS, RSX-11M and RT-11.

Kermit-11 will run on RSX-11M version 4.0 and RSTS/E version 7.2 as long as the task was built without using RMSRES. To be able to build Kermit on RSTS/E version 7.2 of RSX-11 version 4.0 you will have to get RMSLIB.OLB and MAC.TSK from RSX-11 V4.1 or RSTS/E V8.0. The need for version 2 of RMSLIB is due to the use of $SEARCH, $PARSE, $RENAME and $DELETE. The need for the newest MAC.TSK is due to the use of new directives such as .SAVE, .RESTORE and .INCLUDE /FILENAME/.

Note: For RSTS/E system users please note that you do not have to create RMS files as output. You can instead either type set record-format stream, modify the default in K11RMS, MAC or put the set command in one of the following files:

SY:KERMIT.INI
LB:[1,2]KERMIT.INI
SY:[1,2]KERMIT.INI
KERMIT:KERMIT.INI

Restrictions: Minimum System requirements to ASSEMBLE and LINK KERMIT:

RSTS/E V8.0 or later, with multiple private delimiters and RMS V2
RSX-11M V4.1 or later, with full duplex terminal driver and RMS V2
RSX-11M-PLUS V2.1 or later, with full duplex terminal driver and RMS V2. RT V5 will not run under RT-11 SJ. Needs FB or XM.

Minimum system requirements to RUN KERMIT:

RSTS/E V7.2 or later, with multiple private delimiters
RSX-11M V4.0 or later, with full duplex terminal driver
RSX-11M-PLUS V2.0 or later, with full duplex terminal driver
RT-11 V4.

Documentation on magnetic media.

Media (Service Charge Code): 600' Magtape (MA)

Format: DOS-11

new
11-746

User Command Linkage-Plus

Version: V6J, July 1984

Author: William K. Walker, Monsanto Research Corp.,
Miamisburg, OH

Operating System: RT-11 V5 and 5.01

Source Language: MACRO-11

Memory Required: 8872 Words

UCL+ is a user command linkage program for use with RT-11 V5 and later monitors (it will also work under TSX+). It allows dynamic, on-line definition of user commands (or "symbols") and is upward-compatible with the UCL program distributed with RT-11 V5.01. Among the extended features are:

1. An "execute-immediate" mode for commands that are defined in terms of other UCL+ commands.

2. Deletion of multiple symbol definitions in a single command line.

3. Optional chaining to additional "UCL's".

4. A user-definable "run-by-name-path" which extends the RT-11 "run-from-SY:" default.

5. Provision to STORE/RECALL program settings to/from a separate ".UCL" file.

6. A PASS-ON command that allows you to force UCL+ to "pass-on" a given command string to the next "UCL" in the chain (the default mode) or to a program that you specify.

7. DISPLAY of command expansions with or without execution. The DISPLAY command can also be used to output pre-defined ASCII strings to the console, the printer, or some other device/file (handy for sneaky escape sequences).

8. Several useful built-in "hard-wired" commands including a DCL-style RNO command for use with the DECUS RUNOFF program.

9. Provision for accepting lower-case input (as well as most control characters).

10. Provision to list all program parameters, including symbol definitions; list output may be directed to devices/files other than the console.

The V6J distribution also includes the source text and instructions necessary to create an on-line HELP facility for UCL+.

Restrictions: UCL+ will run under RT-11 V5 or later monitors. Version 5 must be sysgened for UCL support.

Documentation on magnetic media.

Media (Service Charge Code): Floppy Diskette (KA), 600' Magtape (MA)

Format: RT-11

revision
11-709

DECODE4: RT-11 SAV Files Disassembler

Version: V2, February 1984

Author: Henry O. Peterson, Bend, OR

Operating System: RT-11 V3, Heath HT-11 H101A-5

Source Language: MACRO-11

Memory Required: 5.4KW

Other Software Required: If documentation is reformatted or revised, RUNOFF (DECUS No. 11-530), is required.

Now, even if your computer is the result of a pact between the fortress at Maynard and an entity 1500 km to its west (and slightly left); even if as might be expected its software matured well before the LSI-11 revolution you can more likely, using that software, get away with fitting the decoder to and using it more efficiently with that computer.

No more are your programs at the mercy of unpredictable or underdeveloped foreign systems! Now you can render such a system user friendly (as well as capitalize on it).

DECODE 4.0 is a modified version of DECODE 3.0 (DECUS No. 11-342). The program provides some additional features over version 3.0 at the expense of being initially somewhat awkward to use. DECODE 3.0 may be preferable in some cases.

DECODE 4.0 is intended to allow easier decoding of relatively large .SAV files on a machine with relatively small diskette capacity such as Heath HT-11 hardware running either the Heath-supplied software or running RT-11, version 3. DECODE 4.0 was developed from DECODE 3.0 using the above-mentioned Heath HT-11 system.

Changes and Improvements: Assembles with Heath HT-11 System and Utilities. Uses separate file to supply repetitive command line information. Listing controls allow decoder output to be printed or stored on files in parts. Decoding, with Heath HT-11 System, is easier or possible starting at addresses above or below 1000, octal. ".BLKW" can be replaced with ".WORD 0,..0" equivalents. Operation with System diskette removed is discussed in write-up and if you decide it is somewhat supported, write-up has decoding example, etc.

Restrictions: Has not been checked with LDA files.

Documentation on magnetic media.

Media (Service Charge Code): Manual (EA), Floppy Diskette (KA), 600' Magtape (MA)

Format: RT-11

Keywords: Heath-11, LSI-11 Disassemblers
Operating System Index: RT-11

new
11-741

VLOAD: A Program for RT-11 Extended Memory Monitor

Version: V2, January 1984

Author: Raquel K. Sanborn, Research Corporation of the University of Hawaii, Honolulu, Hawaii

Operating System: RT-11 V4, XM Monitor

Source Language: MACRO-11

Memory Required: 28KW

Other Software Required: RT-11 Extended Memory Monitor.

Special Hardware Required: Memory Management Unit; Extended Memory (more that 28K words)

This is a program for RT-11 Extended Memory Monitor. It's purpose is to run a un-overlaid foreground (or system) virtual job in such a way as to allow it to run with full (or part, programmer selectable) 32K words of memory. As a result, it only takes up 4K words in the background job's space and makes the most of the extended memory.

Note: This program will need to be adapted to each use, and should be installed by someone with MACRO-11 experience. In order to load virtual job as a single step, it must be read into contiguous physical memory. Since the 4K words in the KERNAL mapped space and the .SETTOP extended-memory (user space) are different, you cannot read (and probably write) across this boundry. This is why the file is read

into high memory (above the 4K words and moved down to it's execution address.

Restrictions: RT-11 time of day feature doesn't work. All restrictions of virtual jobs apply; e.g. no interrupt routines; no access to KERNAL Mapped Monitor Memory; I/O page not valid unless explicity mapped.

Documentation on magnetic media.

Media (Service Charge Code): Write-Up (AA), Floppy Diskette (KA), 600' Magtape (MA)

Format: RT-11

Keywords: RT-11 - Utilities
Operating System Index: RT-11

new
11-SP-66

Symposium Tape from the RT-11 Sig, Spring 1984, Cincinnati, OH

Version: Spring 1984

Author: Various

Submitted By: R. W. Barnard, Sandia National Laboratories, Albuquerque, NM

Operating System: RT-11 V5.0, and RT-11 V5.1 for the Professional-350

Source Language: BASIC-11, FORTRAN IV, FORTRAN 77, MACRO-11

Memory Required: Various

Other Software Required: If necessary, it will be specified in each individual program's documentation.

Special Hardware Required: Various (Specified in each individual program's documentation).

The symposium tape from the RT-11 SIG contains fifteen packages. The packaging format is variable-size subdevices; the files TAPE.DIR and README.1ST at the beginning of the tape describe the contents and how to recover them from the tape. The tape contains the following submissions:

1. Benchmarks for comparing RT-11 FORTRAN-77 with FORTRAN-IV.
2. Program to read individual files from a magtape written with the BUP backup utility.
3. Disk librarian (including the sources), which allows on-line cataloging and location of files.
4. Second release of HGRAPH, a 2- and 3-dimensional plotting package for use on Tektronix-compatible termials and various plotters.
5. Program and data files to display natural images (e.g., photographs) on the PRO-350 under RT-11. The program illustrates how to access the graphics bitmap of the PRO.

6. Vector-to-Raster translator for LA100 graphics. The program converts graphics instructions into a "bitmap" file which can be printed on the LA100 in graphics mode.
7. Programs for the control of DZ(V)11 I/O lines under DIBOL.
8. A FORTRAN program line number resequencer which works for both FORTRAN-IV and FORTRAN-77.
9. An RT-11 version of RUNOFF which is an almost complete emulation of DSR (Digital Standard Runoff fot the VAX).
10. An update of the TSX+ (*) system services library for FORTRAN users.
11. Two different UCL (User Command Linkage) programs: one causes keyboard commands to be treated in a way compatible with TSX+, while the other can be used as an enhancemant to the DEC-distributed UXL provided with RT-11 version 5.01.
12. An object file to source file translator. In contrast to one which works on .SAV files, this translator can determine variable names, subroutine names, and .PSECT information.

13. The King James version of the Bible, converted to magnetic media with an optical scanner.
14. "Housekeeping" files, such as a program for RSTS/E users to recover files from the subdevices, and annotated directories of this and previous SIG tapes.

No guarantees are made as to the completeness, useability, or quality of the programs on tape and the material has not been checked or reviewed.

(*) TSX+ is a product of S & H Computer Systems, Inc.

Note: Only one program (IMAGE - RT-11 Natural Display Program), is specific for RT-11 V5.1 on the Professional-350.

Restrictions: If necessary, it will be specified in each individual program's documentation.

Documentation on magnetic media.

Media (Service Charge Code): Write-Up (AA), 2400' Magtape (PS)

Format: RT-11

Keywords: Symposia Tapes - RT-11
Operating System Index: RT-11